# AN ARCHITECTURE FOR THE UNIFRAME RESOURCE DISCOVERY SERVICE

A Thesis

Submitted to the Faculty

of

Purdue University

by

Nanditha Nayani Siram

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2002

| Report Documentation Page | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| 1. REPORT DATE **MAY 2002** | 2. REPORT TYPE | 3. DATES COVERED **00-00-2002 to 00-00-2002** |
| 4. TITLE AND SUBTITLE **An Architecture for the Uniframe Resource Discovery Service** | | 5a. CONTRACT NUMBER |
| | | 5b. GRANT NUMBER |
| | | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | | 5d. PROJECT NUMBER |
| | | 5e. TASK NUMBER |
| | | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Indiana University/Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202** | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSOR/MONITOR’S ACRONYM(S) |
| | | 11. SPONSOR/MONITOR’S REPORT NUMBER(S) |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited** | | |
| 13. SUPPLEMENTARY NOTES | | |
| 14. ABSTRACT | | |
| 15. SUBJECT TERMS | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT **unclassified** | b. ABSTRACT **unclassified** | c. THIS PAGE **unclassified** | **Same as Report (SAR)** | **205** | |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std Z39-18

To Amma, Nana, and Kishan.

## ACKNOWLEDGMENTS

Being a graduate student at the Department of Computer and Information Science, Indiana University-Purdue University-Indianapolis has been a great learning experience for me. The knowledge gained will be valuable to me for the rest of my life.

I would like to take this opportunity to thank the many people who have helped make this thesis possible.

First and foremost, I would like to thank Professor Rajeev R. Raje, my advisor, for his constant encouragement and guidance through the course of my graduate studies and thesis. His immense knowledge and insights provided a strong foundation for this thesis. He constantly challenged me to achieve greater heights and realize my full potential. Dr. Raje, I thank you once again.

My special thanks to Professor Andrew Olson for being on my advisory committee and providing me guidance during critical periods of my thesis. His painstaking efforts to review my thesis and work are greatly appreciated.

I would like to thank Professor Yung-Ping Chien for being on my advisory committee and providing valuable inputs towards the thesis.

Special thanks to my colleague, Muris Ridzal, for his assistance in the testing of the URDS prototype.

I would like to thank the U.S. Department of Defense and the U.S. Office of Naval Research for supporting this research under the award number N00014-01-1-0746.

Many thanks to the faculty, staff and all my colleagues at the Department of Computer and Information Science for their cooperation and goodwill.

Finally, I would like to thank my parents, and my husband Kishan Siram for their love, encouragement and support.

TABLE OF CONTENTS

## LIST OF TABLES

LIST OF FIGURES

Figure Page

Page

## ABSTRACT

Siram, Nanditha Nayani, M.S., Purdue University, May 2002. “An Architecture for the UniFrame Resource Discovery Service”. Major Professor: Rajeev Raje.

The software development for any large-scale, Distributed Computing System (DCS) is a major challenge. One solution to address the design complexity of a DCS is the "UniFrame" approach. UniFrame provides a comprehensive framework unifying existing and emerging distributed component models under a common meta-model that enables the discovery, interoperability, and collaboration of components via generative software techniques. This thesis presents an architecture and implementation for the resource discovery aspect of this framework, called the UniFrame Resource Discovery Service (URDS). The proposed architecture addresses the following issues: a) the dynamic discovery of heterogeneous software components which offer and utilize services, and b) the selection of components meeting the necessary functional as well as non-functional requirements (such as desired levels of QoS Quality of Service). The thesis also compares the URDS architecture with other Resource Discovery Protocols, outlining the gaps that the URDS is trying to bridge. In the URDS the native registries/lookup services of various component models are extended to be ‘active’ (i.e. listen/respond to periodic multicast announcements) and also have introspection capabilities to discover not only the instances but also the specifications of the components registered with them. “Services” in UniFrame are implemented in different component models and described by their UniFrame specification outlining the computational, functional, cooperational, auxiliary attributes and QoS metrics. The URDS security model provides for the authentication of the principals involved, an access control to multicast address resources, and an encryption of data transmitted.

# 1. INTRODUCTION

Component-based software design has been a growing trend in the development of software solutions for distributed systems. The software realization of a distributed computing system (DCS) is typically achieved by using the notions of independently created and deployed components, with public interfaces and private implementations, loosely integrating with one another to form a coalition of distributed software components. Assembling such systems requires either automatic or semi-automatic integration of software components, taking into account the QoS (Quality of Service) constraints advertised by each component and the collection of components. The UniFrame Approach (UA) [RAJ01, RAJ02] provides a framework that allows an interoperation of heterogeneous and distributed software components and incorporates the following key concepts: a) a meta-component model (the Unified Meta Model – UMM [RAJ00]), with a associated hierarchical setup for indicating the contracts and constraints of the components and associated queries for integrating a distributed system, b) an integration of the QoS at the individual component and distributed application levels, c) the validation and assurance of the QoS, based on the concept of event grammars, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available component choices. The UniFrame approach depends on the discovery of independently deployed software components in a networked environment. In this thesis, an architecture and implementation for the resource discovery aspect of this framework, called the UniFrame Resource Discovery Service (URDS) is described. The URDS architecture provides services for an automated discovery and the selection of components meeting the necessary QoS requirements specified by a component assembler or system integrator. URDS is designed as a

Discovery Service wherein new services are dynamically discovered while providing component assemblers with a Directory style access to services. The result of using URDS, the UniFrame Approach (UA) and its associated tools is a semi-automatic construction of a distributed system.

## 1.1 Problem Definition and Motivation

There has been an increase in the development of technologies for the dynamic discovery of resources such as printers, mail-boxes, memory space and disk space that are available in every network. The growth in the popularity of portable devices such as laptops, PDAs, and cell phones which require configuration each time they attach to a new network segment has also sparked the need for developing protocols which facilitate spontaneous discovery and enable such devices to connect to these resources. However, research in the area of dynamic discovery has been focused solely in the area of discovering and configuring "devices".

There is a need to bring about a change in paradigm from dynamic discovery being purely "device" based to a "service" based approach. "Service" here refers to applications developed using different distributed computing models, with public interfaces and private implementation, that perform computation or actions on behalf of client users. Several directory-based approaches exist for discovering services. These directory-based discovery services are built around the publish-subscribe model wherein services publish their interfaces with a central directory and clients discover these services by contacting the directories. Prominent among these are Universal Description, Discovery and Integration (UDDI) registry [UDD00], CORBA Trader Services [OMG 00], Java$^{TM}$ Remote Method Invocation (RMI) [ORF97], Distributed Component Object Model (DCOM$^{TM}$) [MIC98], etc. However, almost all of these directory/registry services do not assume the presence of other models. The interoperability, which they provide, is limited mainly to the underlying hardware platform, operating system and/or implementational languages. This defeats the underlying goal of discovery being universal.

The problem that needs to be solved is to provide for a “service” discovery model that is dynamic and encompasses services developed in diverse distributed computing models.

The motivation for creating a universal dynamic resource discovery service is as follows: The software systems in any organization constantly undergo changes and evolutions. Moreover, these organizations may be geographically (or logically) dispersed necessitating a communication between independently created and deployed components, loosely integrating with one another to form a coalition of distributed software components. In order to deal with the constant evolutions and changes in the software systems, there is a need to rapidly create software solutions for distributed environments using a component-based software development approach. The solution of decreeing a Common Off The Shelf (COTS) environment, in an organization, will require creating an ensemble of heterogeneous components, each adhering to some model. If reliable software needs to be created for a DCS by combining components, then the quality of service offered by each component needs to become a central theme of the software development approach. Assembling such systems requires either automatic or semi-automatic integration of software components.

The key to automating the process of assembling DCS is to have an infrastructure, that allows for a seamless integration of different component models and sustains cooperation among heterogeneous components. Such an infrastructure will need to provide services to dynamically discover the presence of new components in the search space which offer and utilize services, and allow for the selection of components meeting the necessary functional as well as non-functional requirements (such as desired levels of Quality of Service). The infrastructure will also need to provide translation capabilities for specific models. The UniFrame Resource Discovery Service (URDS) architecture, proposed in this thesis, is designed to provide this infrastructure to support universal dynamic resource discovery.

## 1.2 Objectives: Statement of Goals

The objectives of this thesis are:

- To bring about a shift in paradigms in the field of spontaneous resource discovery from a "device" based discovery approach to a "service" based discovery approach. Traditionally the field of dynamic resource discovery has been oriented towards spontaneous discovery of "devices". This thesis aims to use discovery techniques to detect "services" developed in diverse component models. The thesis also aims to propose a framework wherein the dynamic resource discovery service will be coupled with a container for adapters to provide integration broker capabilities.

- To propose an architecture structured as a platform independent design specification for the URDS. The architecture will address the issues of scalability, adoptability, security and fault tolerance and aim to provide an encompassing view of the end service, computation and middle support required. The architecture will provide details of the information specifications, configurations of the interacting components including detailed algorithms of the functions required to support distributed interaction between objects and an analysis of these algorithms.

- Develop a proof-of-concept prototype for the URDS and validate the principles behind URDS.

- Indicate the inter-relations between the URDS and other components of UniFrame.

## 1.3 Contributions of this thesis

The contributions of this thesis are:

- Provides a comprehensive survey of the existing resource discovery services detailing their characteristics. The survey outlines the purpose of existing resource discovery services and establishes the need for the development of a resource discovery service which is capable of bridging the interoperability gap that is inherent is current discovery services.

- Proposes a novel approach to discovering services dynamically based on the concept of extending the native registries of different component models to be 'active' and 'introspective'. The architecture proposed also automates the process of selecting components based on their computational, cooperational, auxillary attributes and QoS metrics.

- Proposes a framework wherein the dynamic resource discovery service can be coupled with a container for adapters to provide integration broker capabilities.

- Proposes a platform independent architectural model for the UniFrame Resource Discovery Service. The following aspects of the URDS architectural model are presented at appropriate levels of abstraction:
    - *Information Aspect* describing the scope and nature of information specifications.
    - *Computational Aspect* describing the configurations of the interacting computational objects.

- *Engineering Aspect* wherein the mechanisms and functions required to support distributed interaction between objects is described.
- *Technology Aspect* wherein a detailed description of the components and the technology used to realize the system is presented.

## 1.4 Organization of this thesis

This thesis is organized into 6 chapters. The introduction of the problem domain, problem statement, definition of goals and the contributions of the thesis were presented in this chapter. Chapter 2 provides a survey of the existing resource discovery services and establishes the need for a meta-level "service" based discovery service. Chapter 3 provides an overview of UA and URDS and outlines the design concepts on which the URDS architecture is based. Chapter 4 describes the URDS architecture focusing on the high-level design, and the algorithms and interactions of the components that comprise the architecture. Chapter 5 describes a prototype implementation and its validation through experimentation. Finally, this thesis concludes with a discussion of the features of URDS in comparison with other resource discovery services in Chapter 6.

# 2. BACKGROUND AND RELATED WORK

This chapter provides a survey of existing resource discovery protocols under the categories of directory-based and discovery-based services. Section 2.1.1 describes various directory services. Section 2.1.2 describes various discovery services. A comparison between the features of directory-based and discovery-based resource discovery services is presented in Section 2.1.3.

## 2.1 Resource Discovery Protocols

Resource Discovery Protocols perform the task of identifying resources on a network and making these resources accessible to users and applications. Resources may include various services such as web services, component services, or devices such as conventional computers, hand held computers (PDAs), peripheral devices such as printers, and specialized network devices such as digital cameras, and telephones. The protocols for resource discovery are presented under the following two categories:

- Lookup (Directory) Services and Static Registries.
- Discovery Services.

The primary difference between 'discovery' and 'lookup' services is that a discovery service usually supports lookup, but many 'lookup' services do not support discovery [MCG00]. A detailed discussion of these categories is presented in the following subsections.

### 2.1.1 Lookup (Directory) Services and Static Registries

Lookup as defined by McGrath in [MCG00] refers to the *"process of locating a specific object or resource either by exact name or address, or by some matching criteria. Lookup is 'passive', in that it is initiated by a seeker, and requires the existence of some directory or other agent to answer the request. Lookup may be done in a statically configured environment; the directory need not be writable"*. Some examples of lookup services include Universal Description, Discovery and Integration (UDDI) registry [UDD00], CORBA Naming and Trader Services [OMG00, OMG01], Light Weight Directory Access Protocol (LDAP) [WAH97], Domain Name Service (DNS) [MOC87], X.500 [COU01] and GNS [COU01]. The following subsections provide a brief discussion of the above-mentioned 'lookup' services.

#### 2.1.1.1 Universal Description, Discovery and Integration (UDDI) Registry

Universal Description, Discovery and Integration (UDDI) as defined in [UDD00] is a *"specification for distributed Web-based information registries of Web services i.e., UDDI specifications define a way to publish and discover information about Web services."* The term "Web service" is defined in [UDD00] as - *"A Web Service describes specific business functionality exposed by a company, usually through an Internet connection, for the purpose of providing a way for another company or software program to use the service"*. Since UDDI aims at providing an open specification and set of tools for discovering Web Services on the Internet it utilizes the World Wide Web Consortium (W3C) [W3C] standard service definition language - Web Services Description Language (WSDL) [CHR01]. WSDL is an XML [BRA00] grammar for describing the capabilities and technical details of Simple Object Access Protocol (SOAP) [BOX00]-based web services. UDDI data is hosted by operator nodes, which are companies that are committed to running a public node that confirms to the specifications governed by the uddi.org consortium. Three public nodes exist which include Microsoft, IBM and HP whose contents are synchronized regularly.

The components involved in the UDDI architecture are the Web Service Providers, Service Requesters and Service Brokers (UDDI Business Registries).
The fundamental operations performed by these components are:

- Service providers deploy and publish services by registering them with the Service Broker. The interaction with the Service Broker and the registration process is done using the UDDI API. The UDDI APIs are in the form of SOAP-based web services that fall into two categories, namely inquiry and publishing. To invoke these APIs SOAP message is sent with the appropriate body content.

  ```
  Example: <find_business generic='1.0' xmlns='urn:uddi-org:api'>
                <name>XYZ industries </name>
           </find_business>
  ```

  The SOAP response contains all businesses that match criteria and registered services for each business in the form of XML data structure.

- Service requestors find services by searching the Service broker's registry of published services. Locating services is done via a combination of UDDI and Web Services Description Language (WSDL).

- Service requestors bind to the Service provider and consume the available services. Binding to Service Providers leverages information specified in WSDL and the SOAP protocol.

#### 2.1.1.2 CORBA Trader Services

The CORBA Trader Service [OMG00] provides "Yellow Pages" for objects, enabling them to publicize/request services. The Trader Service facilitates 'matchmaking' between service providers (*Exporters*) and service consumers (*Importers*). The exporters, which are CORBA objects, use the interfaces provided by the Trader Services to register their services. The information about the service registered with the Trader comprises of a reference to the object providing the service, information about the operations supported such as their names, parameters and result types, and other properties describing the capabilities of the service.  The Trader stores the type descriptions in a repository and also maintains a database of service objects. The clients or importers, which are CORBA Objects, can query the Trader for a list of services registered with it or make a request for a particular service by specifying the service type and properties desired in the service. The trader will find a match for the client based on the search criteria. Traders can be linked to form a *federation of traders*. This linkage of traders allows a trader to make the offer spaces of other traders become implicitly available to its own clients. The Trader Service does not guarantee that the registered objects are available. It also does not provide notification and security features and needs to be augmented with the CORBA event services and security services to provide these features. The Trader Service is defined as CORBA interfaces, and all advertisements, requests, and replies are CORBA objects. Since the Trader Services depends on CORBA, all participants must be cast as CORBA objects and use CORBA protocols.

#### 2.1.1.3 X.500 and Lightweight Directory Access Protocol (LDAP)

LDAP (Lightweight Directory Access Protocol) is a software protocol for enabling anyone to locate organizations, individuals, and other resources such as files and devices in a network, whether on the public Internet or on a corporate intranet. LDAP is a "lightweight" (smaller amount of code) version of Directory Access Protocol (DAP), which is part of X.500 [ITU97], a standard for directory services in a network. The directory service is organized like a tree and is referred to as a Directory Information Tree (DIT)[WIL99]. The LDAP data format is structured as a scalable hierarchy of name spaces, with an added flexibility to form a relational grouping of *entries*. An entry in the LDAP server is analogous to a record in a relational database. LDAP is extensible, and allows storage of different kinds of information. LDAP' s query model allows for search and retrieval of entries stored in the LDAP directory server. Since LDAP is organized as a hierarchy, the queries can be limited to particular parts of the hierarchy. Though the design is scalable queries over very large domains are likely to be very inefficient. LDAP does not have any built in security model and relies on other network services for this purpose. LDAP does not specify protocols for "spontaneous" discovery and because of the complexity; it may not be well suited for either near real-time discovery, or for very large numbers of services [MCG00].

#### 2.1.1.4 Global Name Service (GNS) and Domain Name Service (DNS)

*Global Name Service* [COU01] is designed to provide facilities for resource location, mail addressing and authentication. GNS manages a naming database that is composed of a tree of directories holding names and values.

*Domain Name System* (or *Service*), [MOC87], is an Internet service that translates domain names into IP addresses. The DNS protocol provides static database of name-address maps, which is hierarchically partitioned. The naming data is replicated and cached in order to achieve scalability. Recent extensions to DNS support a very limited set of service types and a few attributes that can be used to search [GUL96]. The types of queries supported by DNS include host name resolution and reverse resolution, mail host location, host information and well-known services information [COU01]. DNS is a trusted service, security is provided by controlling access to a few privileged users. Arbitrary user applications may not add or modify the DNS database.

### 2.1.2 Discovery Services

"Discovery" refers to the spontaneous process, in which resources or services 'discover' other resources on the network, and present themselves to these other resources.

Several protocols exist for service discovery, prominent among which are: Service Location Protocol (SLP) [GUT99b], JINI [SUN01a], Ninja Project: Secure Service Discovery Service (SSDS) [CZE99, NIN02], Salutation [SAL, SAL99a, SAL99b], Bluetooth [BLU], Universal Plug and Play (UPnP) [CHR99, UPNP99], and Simple Service Discovery Protocol (SSDP) [GOL99].

The following sections provide an overview of the above-mentioned discovery services.

### 2.1.2.1 Service Location Protocol (SLP)

SLP Version 2 [GUT99a] is an Internet Engineering Task Force (IETF) standard framework for resource discovery. SLP architecture comprises of the following components:

- *User Agents (UA):* UAs perform service discovery on behalf of clients.
- *Service Agents (SA):* SAs advertise the location and characteristics of services.
- *Directory Agents (DA):* DAs act as directories which aggregate service information received from SAs in their database and respond to service requests from UAs.

DAs can be replicated or organized as hierarchical or graph of domains.

Discovery of DAs by UAs and SAs can be implemented in following configurations:

- *Active Discovery:* SAs and UAs multicast SLP requests.
- *Passive Discovery:* DAs announce their presence by periodically multicasting advertisement messages.
- *Dynamic Host Configuration Protocol (DHCP) Options for SLP* [PER99]: UAs and SAs can locate DAs using DHCP wherein DHCP servers configured with this option distribute the DA addresses to agents that require them.

SLP configuration allows the following two modes of operation [GOV00]:

- *Without DAs:* This requires no administration. UAs multicast or broadcast service requests to SAs, which are listening on well-known ports. When a match is found SAs respond to the UAs using unicast.
- *With DAs:* The protocol is more efficient, in this configuration. The UAs, SAs and DAs are configured as members of a scope and communication takes place between them only if they support the same scope. UAs and SAs communicate with DAs, which are listening on well-known ports using unicast messages.

Services are advertised using a “Service URL” [GUT99a] and a “Service Template”[GUT99c]. The Service URL comprises of the IP address of the service, the port number, and path. Service Templates specify the attributes that characterize the service and their default values. Service requests may match according to service type or by attributes. SLP supports fairly powerful syntax for attribute matching based on templates and LDAPv3 predicate [WAH97]. SLP supports authentication, but does not specify it. It does not support encryption.

### 2.1.2.2 JINI

JINI is Java based framework for spontaneous discovery developed by Sun Microsystems. [EDW99] JINI architecture [SUN01a] is similar to SLP. However, JINI is very tightly bound to the Java environment. Each JINI device is assumed to have a Java Virtual Machine (JVM) running on it. JINI uses Java Remote Method Invocation (RMI) [RMI] protocol for communication, which supports exchange of serialized Java objects. The main components of a JINI system and their functions are:

- *Service:* A service is a logical concept defined and identified by a Java interface. The publicly visible part of the service, which is downloaded by the clients, is referred to as the “service object” or “service proxy”. A service registers this “service object” or “service proxy” with the Lookup service.

- *Client:* The client contacts the Lookup Service requesting a service and the Lookup service returns it the “service object” or “service proxy” which is copied to the client’s JVM and any further communication with the service by the client is conducted through the proxy. The client request is basically a simple template for matching string attributes. However, the request and the matching must be implemented as Java objects, following JINI specified interfaces.

- *Lookup Service (Service Locator):* The Lookup Service serves as a directory of available services. Lookup Service listens on a well-known port for unicast or multicast messages. When Lookup service gets requests from services to register them or from clients requesting services, it returns them a “registrar” object, which serves as a proxy to the Lookup Service and runs on the service’s or client’s JVM. Any requests, which are to be made of the Lookup service, can be made through the proxy “registrar”.

The JINI Discovery protocol is used by JINI components (services, clients, lookup services) to locate other relevant JINI components in the network. The discovery process operates in the following two modes:

- *Multicast:* The discovery processes, in this mode are categorized as *aggressive, lazy* and *peer lookup*. During *aggressive discovery* a JINI component transmits probes at a fixed interval for a specified period, or until it has discovered a sufficient number of Lookup services. Aggressive discovery only happens at component initiation time. After completion of aggressive discovery the component enters *lazy discovery*, where it listens for announcements sent at intervals by lookup services. During lazy discovery a lookup service both listens for announcements by other lookup services and sends its own announcements at the required intervals. *Peer Lookup* [GOV00] is mentioned in the JINI specification as a technique used by clients to discover services in the absence of a lookup service. Clients multicast request packets and they receive direct response from the services from which the clients can select the services they are interested in and drop the rest.

- *Directed:* In this mode the components use a directed discovery process in which the components have a list of lookup services to discover with which they try to establish connection.

Other features of the JINI architecture include the concept of *Leasing, Remote Event Notifications* and *Transactions*. *Leasing* mandates that all advertisements and registrations are valid only for a specific period of time thus forcing all clients and services to renew their leases periodically. Leases ensure that JINI recovers from crashed entities as these are automatically purged from the lookup services when the lease expires and periodic renewal of leases by entities rebuilds the global state in case of a Lookup Server failure. *Remote Event Notifications* allows an object to transmit notifications of events that have occurred within the object to other objects, which may have registered their interest in such events. *JINI Transactions* provide for a set of wrapped operations wherein an entire set of operations succeeds or fails without scope for partial success.

Some of the drawbacks of the JINI architecture include a limited filtering mechanism as when compared to other services like LDAP, SLP, CORBA Trader Service, or other XML based approaches. JINI cannot interoperate with any other protocol or language environment. The devices must either implement a JVM, or else use a proxy. JINI security is based on the weak security provided by Java and cryptography is not mandatory.

#### 2.1.2.3 Ninja Project: Secure Service Discovery Service (SSDS)

The SSDS [CZE99, NIN02] is part of the Ninja research project at University of California, Berkeley. The main components of the SSDS system and their function in the discovery process is as follows:

- *Service Discovery Service (SDS) Servers:* SDS Servers are organized into hierarchical domains (domain specifies their network extent). The servers periodically multicast authenticated messages containing the multicast address for sending service announcements. The SDS servers cache the service descriptions that are advertised in the domain.
- *Services:* The Services listen for SDS Server announcements and multicast the service descriptions to the multicast address using authenticated, encrypted, one-way service broadcasts.
- *Clients:* Clients discover the SDS server for their domain by listening to a well-known SDS global multicast address. A client uses Authenticated RMI [CZE99, WEL] to connect to the SDS server and submits a query in the form of an XML template.
- *Certificate Authority (CA):* The SDS uses certificates signed by the CA to authenticate bindings between principals (components in the SDS system) and their public keys.
- *Capability Manager (CM):* The SDS uses capabilities as an access control mechanism to enable services to control the set of users that are allowed to discover their existence. The CM generates and distributes capabilities to all the users.

The SSDS shares similarities with other discovery protocols, with significant improvements in reliability, scalability, and security. SSDS is implemented in Java and uses Java-RMI for remote calls. It uses XML for service description and location.
The protocol is designed as an exchange of XML "documents", and service location is implemented by matching of XML tags [CZE99, HOD99]. SSDS provides extremely strong mandatory security: all parties are authenticated, and all message traffic is encrypted. The security features supported include: assurance of authenticity of discovery service, assurance of the privacy and authenticity of service descriptions, two-way authenticated and encrypted remote method invocation, and capabilities to authenticate

all principals. The hierarchical organizations of the SDS Servers increases system scalability and also serves for failure detection and automatic restart of failed servers.

#### 2.1.2.4 Salutation

The Salutation protocol [SAL, SAL99a, SAL99b] is an open specification that provides "spontaneous" configuration of network devices and services. The Salutation architecture defines an abstract model with three components: Client, Server, and Salutation Manager (SLM). The Salutation Manager manages all communication, and bridges across different communication media as needed. Salutation defines its protocol based on SunRPC. Salutation defines a specific (extensible) record format for describing and locating services. This format includes service type (such as ' [PRINT]') and attributes (such as ' color'). Services advertise by registering with one or more Salutation Managers. Clients locate services by sending service requests.

#### 2.1.2.5 Universal Plug and Play (UPnP) and Simple Service Discovery Protocol (SSDP)

UPnP [CHR99, UPNP99] is a Microsoft standard for spontaneous configuration. UPnP handles network address resolution, and coupled with the IETF proposal Simple Service Discovery Protocol [GOL99], it provides higher-level service discovery. UPnP has a similar architecture to Salutation and SLP, and was influenced by them. UpnP uses XML for device/service description and queries, which brings it into the mainstream of the evolving WWW. At the base, UPnP provides "simple discovery", in which network addresses are discovered. Advertisement is done by a local broadcast announcement. When successful, "simple discovery" returns an IP address or URL plus a "device type". Services are described by extended URLs, similar to (but completely incompatible with) SLP. The URL is for an XML file with an elaborate description of the device. Starting with this URL, the SSDP defines a Web based discovery protocol, which uses HTTP (with extensions).

#### 2.1.2.6 Bluetooth Service Discovery Protocol

Bluetooth [BLU] is a new short-range wireless transmission protocol. The Bluetooth protocol stack contains the Service Discovery Protocol (SDP), which is used to locate services provided by a Bluetooth device. It is based on the Piano platform by Motorola and has been modified to suit the dynamic nature of ad hoc communications.

### 2.1.3 Features of Resource Discovery Protocols

The underlying problem that all the resource discovery protocols are aiming solve and the solutions to achieve the same can be broadly summarized as:

- *Service Advertisement:* Service Providers advertise their services by providing their address and other necessary information. This advertisement could be facilitated either by registration with a directory service or through multicast or broadcast communication.
- *Service Request:* Service Consumers seeking services forward their request to some centralized directory service or make known their request to the world again through the process of multicast or broadcast communication.
- *Match Making:* The process in which service producers and consumers hook up. This could again be facilitated through a directory service serving as the intermediary or a direct discovery between producers and consumers. In the case where the directory service serves as the intermediary. The discovery of the service providers could again be spontaneous or through the registration process.

Issues such as reliability and scalability are handled either through a hierarchical or federated organization.  A comparison of the features of the Directory and Discovery services is provided in the table 2.1 compiled based on the characteristics summarization in [MCG00].

| **Feature List** | **Directory Services** | **Discovery Services** |
|---|---|---|
| Information Storage and Retrieval | Storage of information in static databases, which can be replicated for greater scalability. | Spontaneous discovery and configuration of network services and devices. Some Discovery services like SSDS, JINI cache the service information in which case they periodically update the cache to reflect the global state of the system. |
| Failure Detection | Do not monitor the resources for availability or failure due to external circumstances such and node/link failure, etc. They usually do not possess any event generation mechanisms either to inform clients of resource registration or withdrawal. | Automatically configure according to service availability. |
| Management | Centralized control. Usually maintained by privileged administrators. | Decentralized management with limited administration. |
| Search Semantics | Usually provide lower flexibility with respect the search criteria that can be specified. | Allows for selection of very specific types of service. |
| Interoperability | No interoperability between different models or services. | Some services offer interoperability through the |

| | | |
|---|---|---|
| | | use of bridges and proxies. |

Table 2.1. Comparison of the Features of the Directory and Discovery services

The Directory and Discovery Services described this chapter are mostly designed for ‘closed’ systems, i.e., systems, although distributed in nature, are developed and deployed in a confined setup. Such systems do not take advantage of the heterogeneity, local autonomy and the open architecture that are characteristic of DCS. The URDS architecture on the other hand is designed for ‘open’ systems by providing for the discovery and interoperation of distributed heterogeneous software components. These systems can be extended as new needs surface.

Chapter 3 provides an overview of the UniFrame Approach and the UniFrame Resource Discovery Service.

# 3. OVERVIEW OF UNIFRAME APPROACH (UA) AND UNIFRAME RESOURCE DISCOVERY SERVICE (URDS)

Chapter 2 provided descriptions of various resource discovery services under the categories of directory and discovery services. This chapter provides an overview of the UniFrame Approach (UA) and how it can be used for developing DCS. The chapter also presents a brief discussion of the UniFrame Resource Discovery Service (URDS) architecture outlining the design concepts on which the architecture is based. The detailed description of the URDS architecture is presented in Chapter 4.

## 3.1 UniFrame Approach (UA) and UniFrame Resource Discovery Service (URDS)

The UniFrame Approach (UA) aims at facilitating an automatic or semi-automatic creation of a DCS based on an integration of heterogeneous components. The UA specifies a framework for component developers to create, test and verify from the point of view of QoS, the components they develop and deploy on the network and for component assemblers/system integrators to select and semi-automatically generate a software solution for the DCS under consideration.

As indicated in Chapter 1, the Unified Meta-Component Model (UMM) proposed in [RAJ00] is the central part of the UniFrame Approach. The following subsections describe this meta-model.

## 3.2 Unified Meta-Component Model (UMM)

UMM consists of three entities: a) Components, b) Services and Service Guarantees, and c) Infrastructure. URDS represents the infrastructural part of the UMM.

### 3.2.1 Components

Components in UniFrame are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to some distributed-component model but there is no notion of a unified implementational framework. Each component has a state, an identity, a behavior, well-defined interfaces and a private implementation. In addition, each component in UMM has three aspects:

- *Computational Aspect:* The computational aspect reflects the task(s) carried out by each component. It in turn depends upon: a) the objective(s) of the task, b) the techniques used to achieve these objectives, and c) the precise specification of the functionality offered by the component. The computational aspect of a component is described by its *inherent attributes* (book-keeping aspects, e.g., author, version, etc.) and *functional attributes* (formal aspects, i.e., computation, contracts and levels of service).

- *Cooperative Aspect:* The cooperative aspect of a component indicates its interaction with other components. Informally, the cooperative aspect of a component may contain: i) *Expected collaborators* - other components that can potentially cooperate with this component, ii) *Pre-processing collaborators* - other components on which this component depends upon, and iii) *Post-processing collaborators* - other components that may depend on this component.

- *Auxiliary Aspect:* In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of a DCS. The auxiliary aspect of a component addresses these features.

### 3.2.2 Services and Service Guarantees

Services in UniFrame could be a computational effort or an access to underlying resources. In DCS, it is natural to have several choices for obtaining a specific service. Thus, each component, in addition to indicating its functionality, must be able to specify the quality of the service offered.

The Quality of Service (QoS) is an indication given by a software component about its confidence to carry out the required services in spite of its constantly changing execution environment and a possibility of partial failures. The QoS offered by each component is dependent upon the computation performed, algorithm used, expected computational effort and resources required, the cost of each service, and the dynamics of supply and demand. The task of guaranteeing the necessary QoS is a key issue in any quality-oriented framework. The UA to assuring the QoS of a DCS is made up of three steps: a) the creation of a catalog of QoS parameters (or metrics), b) a formal specification of these parameters, and c) a mechanism for ensuring these parameters, both at each individual component level and at the entire system level. In [RAJ02, BRA02], these three steps are described in detail.

### 3.2.3 Infrastructure

URDS is designed to provide the infrastructure necessary for discovering and assembling a collection of components for building a DCS.

The URDS infrastructure (illustrated in Figure 3.1) comprises of the following components: i) Internet Component Broker (ICB) which is a collection of the following services - Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM) ii) Headhunters (HHs), iii) Meta-Repositories, iv) Active-Registries, v) Services (S1..Sn), and vi) Adapter Components (AC1..ACn). Figure 3.1 also illustrates the users (C1..Cn) of the URDS system who can be the Component Assemblers, System developers or System Integrators. Section 3.5 provides a brief overview of each of these components. The numbers in the Figure 3.1 indicate the flow of activities in the URDS. These are explained, in detail, in the context of an example in section 3.5.1.

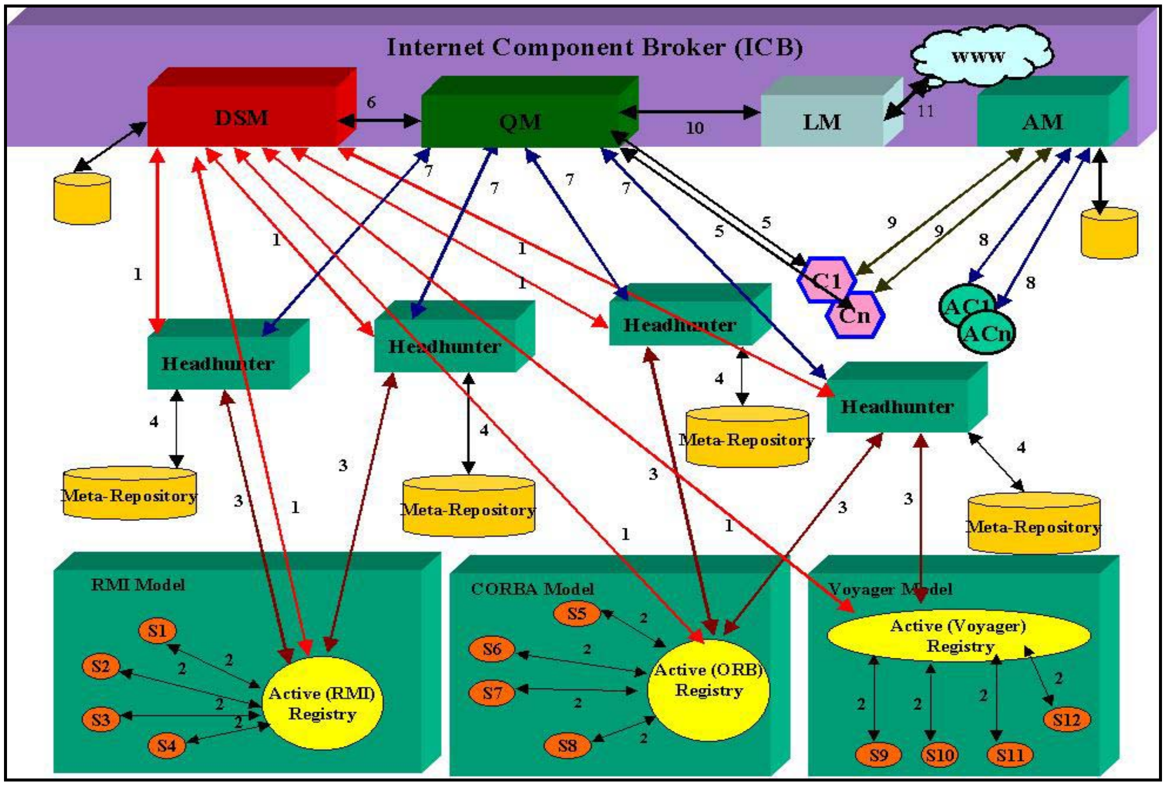


Figure 3.1. URDS Architecture

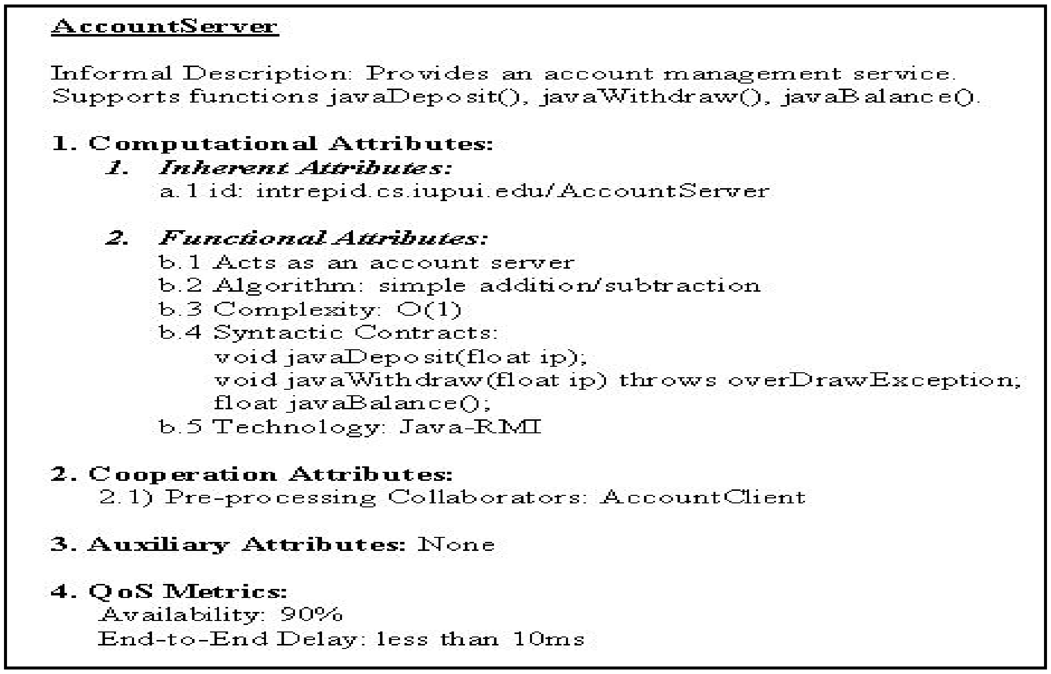
## 3.3 Specification of Components in UniFrame

UniFrame specifications are initially informally indicated in a natural language-like style. This natural-language style specification indicating the computational, cooperative, auxiliary attributes and QoS metrics of the component is refined into a more standard XML-based specification during the Component Development and Deployment Phase as described in Section 3.6.1. XML [BRA00] is selected as the standard for service specification since it is general enough to express the required concepts, it is rigorously specified, and it is universally accepted and deployed.

The XML-based UniFrame Service Specification, which represents the information needed to describe a service, is comprised of:

- ***ID (or Service Type Name)*****:** A unique identifier, comprising of the host name on which the component is running and the name with which this component binds itself to a registry will identify each service.
  Example: intrepid.cs.iupui.edu/AccountServer1
- ***Component Name*****:** The name, with which the service component identifies itself, i.e., the class name of the implementation, which is instantiated at the time of binding. This can be different from the name with which the component binds itself to a registry.
  Example: AccountServer
- ***Description*****:** A brief description of this service component.
  Example: Provides an account management system.
- ***Function Descriptions*****:** A brief description of each of the functions supported by the service component.
  Example: javaDeposit, javaWithdraw, javaBalance
- ***Syntactic Contracts*****:** A definition of the computational signature of the service interface.
  Example: void javaDeposit(float ip), void javaWithdraw() throws OverDrawException, float javaBalance().

- ***Purpose*:** Overall function of the service component.
  Exampe: Acts as an Account Server.
- ***Algorithm*:** The algorithms implemented by this component.
  Example: Simple Addition/Subtraction
- ***Complexity*:** The overall order of complexity of the algorithms implemented by this component. Example: O(1)
- ***Technology*:** The technology used to implement this component.
  Example: CORBA, Java RMI, etc.
- ***QoS Metrics*:** Zero or more *Quality Of Service (QoS) types*. A *QoS type name* defines the QoS value. Associated with a QoS type is the triplet of <*QoS-type-name, measure, value*> where *QoS-type-name* specifies the QoS metric, for example, *throughput*, *capacity*, *end-to-end delay,* etc. *Measure* indicates the quantification parameter for this type-name like *methods completed/sec*, *number of concurrent requests handled*, *time,* etc. *Value* indicates a numeric/string/boolean value for this parameter.
  Example: <Availability,%,90>. A catalog of Quality of Service parameters and their metrics to be used in UniFrame specifications are indicated in [BRA02].



<u>**AccountServer**</u>

Informal Description: Provides an account management service. Supports functions javaDeposit(), javaWithdraw(), javaBalance().

**1. Computational Attributes:**

   ***1. Inherent Attributes:***
      a.1 id: intrepid.cs.iupui.edu/AccountServer

   ***2. Functional Attributes:***
      b.1 Acts as an account server
      b.2 Algorithm: simple addition/subtraction
      b.3 Complexity: O(1)
      b.4 Syntactic Contracts:
```
void javaDeposit(float ip);
void javaWithdraw(float ip) throws overDrawException;
float javaBalance();
```
      b.5 Technology: Java-RMI

**2. Cooperation Attributes:**
   2.1) Pre-processing Collaborators: AccountClient

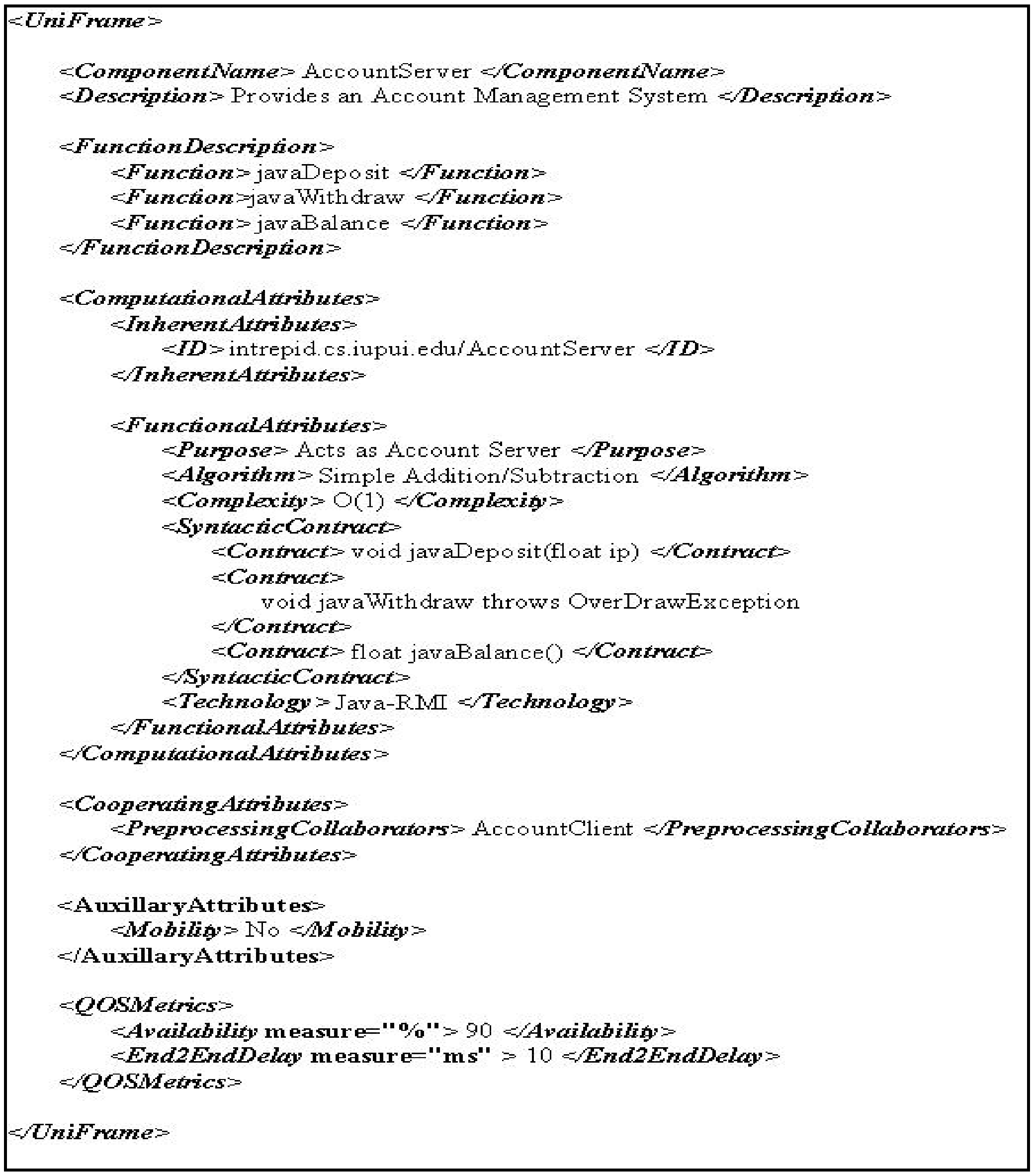
**3. Auxiliary Attributes:** None

**4. QoS Metrics:**
   Availability: 90%
   End-to-End Delay: less than 10ms

Figure 3.2. Example of Informal Natural Language-based Uniframe Specification

Figure 3.2 illustrates an example of an informal natural language-like style UniFrame specification. This example is for a Java-RMI based bank account management system with services for deposit, withdraw, and check balance.

The informal natural language-like specification in Figure 3.2 is translated into an XML-based UniFrame specification illustrated in Figure 3.3.



```
<UniFrame>

    <ComponentName> AccountServer </ComponentName>
    <Description> Provides an Account Management System </Description>

    <FunctionDescription>
        <Function> javaDeposit </Function>
        <Function>javaWithdraw </Function>
        <Function> javaBalance </Function>
    </FunctionDescription>

    <ComputationalAttributes>
        <InherentAttributes>
            <ID> intrepid.cs.iupui.edu/AccountServer </ID>
        </InherentAttributes>

        <FunctionalAttributes>
            <Purpose> Acts as Account Server </Purpose>
            <Algorithm> Simple Addition/Subtraction </Algorithm>
            <Complexity> O(1) </Complexity>
            <SyntacticContract>
                <Contract> void javaDeposit(float ip) </Contract>
                <Contract>
                    void javaWithdraw throws OverDrawException
                </Contract>
                <Contract> float javaBalance() </Contract>
            </SyntacticContract>
            <Technology> Java-RMI </Technology>
        </FunctionalAttributes>
    </ComputationalAttributes>

    <CooperatingAttributes>
        <PreprocessingCollaborators> AccountClient </PreprocessingCollaborators>
    </CooperatingAttributes>

    <AuxillaryAttributes>
        <Mobility> No </Mobility>
    </AuxillaryAttributes>

    <QOSMetrics>
        <Availability measure="%"> 90 </Availability>
        <End2EndDelay measure="ms" > 10 </End2EndDelay>
    </QOSMetrics>

</UniFrame>
```

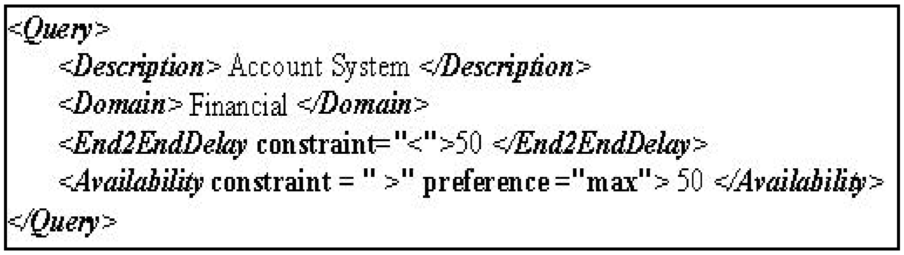
Figure 3.3. Example of translated XML-based UniFrame Specification

Various aspects in the service specification can be identified through the nodes in the XML specification. Nodes can be single valued or multi-valued in which case they can hold multiple child nodes. For example; the node *ComponentName* which introduces the name of the service component is a single valued node, whereas the node *FunctionDescription* which introduces the names of the functions supported by the service component has multiple child nodes each named *Function*.

## 3.4 Query and Description of Target Code Architecture

The UniFrame approach allows the system developers/components assemblers/system integrators to present a query to the system in natural language-like style. This query is passed through a language query processor, which extracts the key words, constraints, and preferences from the query and applies a knowledge base of the domain maintained with it, to the query to construct an XML-based query. The XML-based query is further processed into a structured query language (SQL) statement during the process of matchmaking.

The general form of a query is a request to create a system that has certain QoS parameters. The name of the system is important in identifying the application domain and the QoS parameters help identify desired properties of the system. A sample query can be stated informally as: *Create an account management system that has availability>50% and end-to-end delay < 50ms with preference maximum availability*. (The preference can be specified to indicate that if there are multiple solutions satisfying the requirements, then the search results returned to the developer need to be sorted in the order of greatest availability). The natural language-like query is processed into an XML-based query as depicted in Figure 3.4



```
<Query>
    <Description> Account System </Description>
    <Domain> Financial </Domain>
    <End2EndDelay constraint="<">50 </End2EndDelay>
    <Availability constraint = " >" preference ="max"> 50 </Availability>
</Query>
```

Figure 3.4. Example of a Processed XML-based Query

During matchmaking the above XML based query is further translated into the following structured query language (SQL) statement: SELECT * FROM *componentTable* A, *functionTable* B WHERE (A.ID = B.ID) AND ((*description* LIKE %account%) OR (*description* LIKE %system%)) AND (*end2endDelay* < 50) AND (*availability* > 50).

The above SQL statement is specific to the implementation. The URDS architecture proposes that the UniFrame specification be stored in a database (Meta-Repository) so that the natural language-like query can be translated into a structured query language statement and executed against the tables of the database to find a match for the query. In the sample SQL statement shown the terms *componentTable, functionTable* refer to the database tables in which the parsed UniFrame specification is stored. The *componentTable* maintains the details associated with the component name, description, computational, cooperating, auxillary attributes, and QoS metrics. The *functionTable* holds details associated with function names and their associated syntactic contracts. The terms *description, end2endDelay* and *availability* correspond to the column names of the *componentTable.* The implementation specific details of terms are explained in subsection 5.2.4.1.1.3 of chapter 5.

## 3.5 Overview of the URDS Architecture

The URDS architecture is organized as a federated hierarchy in order to achieve scalability. Figure 3.5 illustrates this hierarchical organization. Every ICB has a one level hierarchy of zero or more Headhunters attached to it. The ICBs in turn are linked together with unidirectional links to form a directed graph. The URDS discovery process is "administratively scoped", i.e., it locates services within an administratively defined logical domain. '*Domain'* in UniFrame refers to industry specific markets such as Financial Services, Health Care Services, Manufacturing Services, etc. The domains supported are determined by the organizations providing the URDS service.

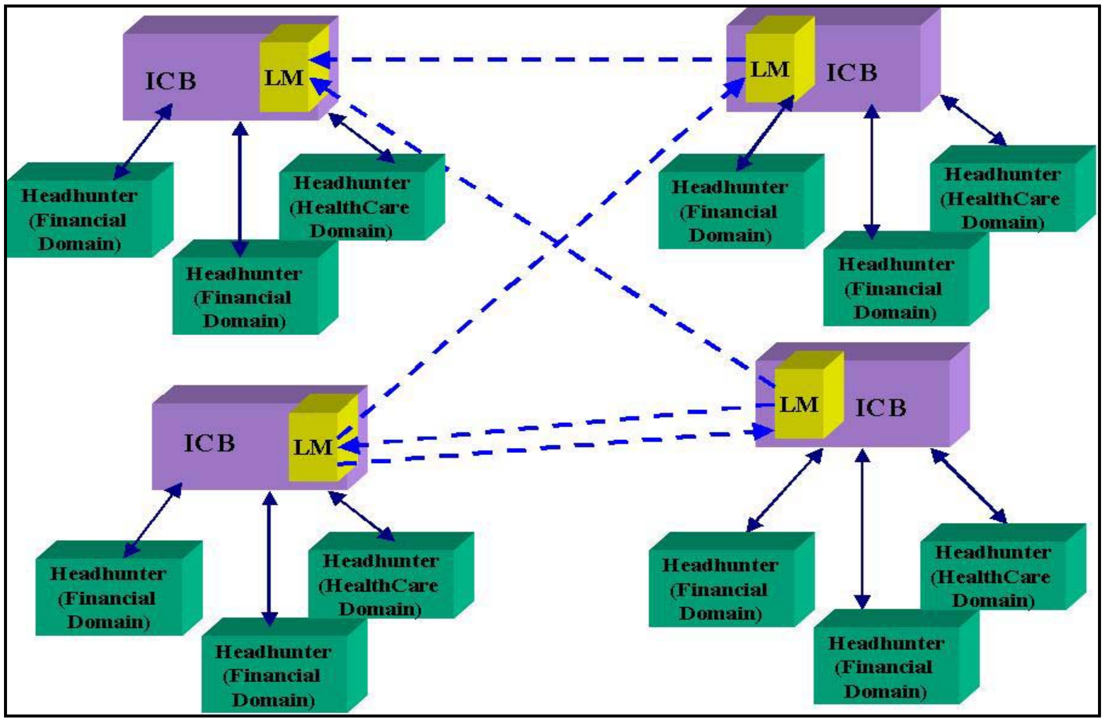

ICB
LM
Headhunter (Financial Domain)
Headhunter (HealthCare Domain)
Headhunter (Financial Domain)
LM
ICB
Headhunter (Financial Domain)
Headhunter (HealthCare Domain)
Headhunter (Financial Domain)
ICB
LM
Headhunter (Financial Domain)
Headhunter (HealthCare Domain)
Headhunter (Financial Domain)
LM
ICB
Headhunter (Financial Domain)
Headhunter (HealthCare Domain)
Headhunter (Financial Domain)

Figure 3.5. Federated Hierarchical Organization of ICBs

The URDS discovery protocol is based on periodic multicast announcements. The multicast communication is subject to various security threats, such as eavesdropping, uncontrolled group access and masquerading. URDS addresses the security threats faced in the scenario of a discovery process. The security model of URDS provides for authentication of the principals involved, an access control to multicast address resources, and an encryption of the data transmitted. The model does not at this stage provide for elaborate protocols for establishing keys, passwords, etc. These are considered as future enhancements and are mentioned in Chapter 6.

The URDS architecture is designed to handle failures through periodic announcements (in case of Headhunters), ‘heartbeat’ probes (in case of Link Managers) and information caching. A lack of communication from the recipients of the communication beyond a designated time range is deemed as a failure of that component and the state of the system is accordingly reset. The caches of the Headhunter and Link Manager are updated based on the responses received from ActiveRegistries and Link Managers in other ICBs respectively or purged based on lack of them.

Table 3.1 gives a brief description of each of the components that comprises the URDS architecture. The elaborate details are provided in the next chapter.

| | |
|---|---|
| **Internet Component Broker (ICB)** | The ICB acts as an all-pervasive component broker in an interconnected environment. It encompasses the communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between heterogeneous components. The ICB is not a single component, but a collection of services comprising of the Query Manager (QM), the Domain Security Manager (DSM), Adapter Manager (AM), and the Link Manager (LM). These services are reachable at well-known addresses. It is envisioned that there will be a fixed number of ICBs deployed at well-known locations hosted by corporations or organizations supporting this initiative. |
| **Domain Security Manager (DSM)** | The DSM serves as an authorized third party that handles the secret key generation and distribution and enforces group memberships and access controls to multicast resources through authentication and use of access control lists (ACL). DSM has an associated repository (database) of valid users, passwords, multicast address resources and domains. |
| **Query Manager (QM)** | The purpose of the QM is to translate a system integrator’s natural language-like query into a structured query language statement and dispatch this query to the ‘appropriate’ Headhunters, which return the |

| | |
|---|---|
| | list of service provider components matching these search criteria expressed in the query. ‘Appropriate’ is determined by the domain of the query. Requests for service components belonging to a specific domain will be dispatched to Headhunters belonging to that domain. The QM, in conjunction with the LM, is also responsible for propagating the queries to other linked ICBs. |
| **Link Manager (LM)** | The LM serves to establish links with other ICBs for the purpose of federation and to propagate queries received from the QM to the linked ICBs. The LM is configured by an ICB administrator with the location information of LMs of other ICBs with which links are to be established. |
| **Adapter Manager (AM)** | The AM serves as a registry/lookup service for clients seeking adapter components. The adapter components register with the AM and while doing so they indicate their specialization, i.e., which component models they can bridge efficiently. Clients contact the AM to search for adapter components matching their needs. |
| **Headhunter (HH)** | The Headhunters perform the following tasks: a) Service Discovery: detect the presence of service providers (Exporters), b) register the functionality of these service providers, and c) return a list of service providers to the ICB that matches the requirements of the component assemblers/system integrators requests forwarded by the QM. The service discovery process performs the search based on multicasting. |
| **Meta-Repository (MR)** | The Meta-Repository is a data store that serves a Headhunter to hold the UniFrame specification information of exporters adhering to different models. The repository is implemented as a standard relational database. |
| **Active Registry (AR)** | The native registries/lookup services of various component models (RMI, CORBA, Voyager) are extended to be able to listen and respond to multicast messages from the Headhunters and also have introspection capabilities to discover not only the instances, but also the specifications of the components registered with them. |

| | |
|---|---|
| **S1..Sn** | Services implemented in different component models (RMI, CORBA, etc.,) identified by the service type name and the component's informal UniFrame specification which is an XML specification outlining the computational, functional, cooperational and auxiliary attributes of the component and zero or more QoS metrics for the component. |
| **AC1..ACn** | Adapter components, which serve as bridges between components implemented in diverse models. |
| **C1..Cn** | Component Assemblers, System Integrators, System Developers searching for services matching certain functional and non-functional requirements. |

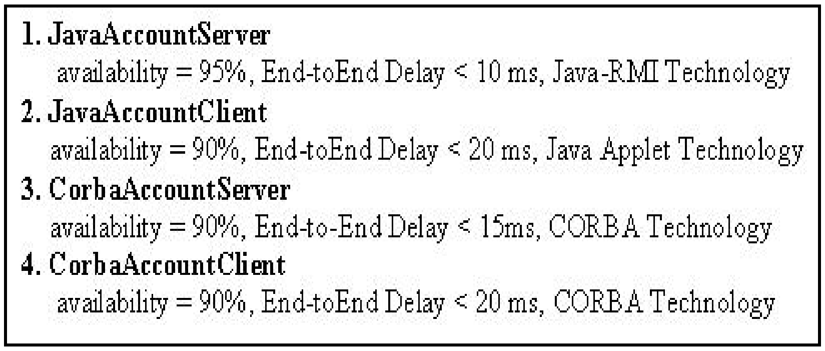
Table 3.1. Description of URDS components

### 3.5.1 Interaction of URDS Components

Table 3.2, outlines the interactions between the URDS components in servicing a query for assembling an account management system. This example assumes the presence of a pair of Java RMI server and Java RMI client programs, and a pair of CORBA server and CORBA client programs, which are available to construct the account management system. The rows of the table are numbered corresponding to the flow of control shown in Figure 3.1. The result of this interaction will be an ensemble of components, which may be assembled into a complete system as described in Section 3.6.2.

| | |
|---|---|
| **1** | This indicates the interactions between the principals (Headhunters/Active Registries) and the DSM.<br>The principals contact the DSM with their authentication credentials in order to obtain the secret key and the multicast address for the group communication (many to one interaction).<br>Example credentials:<br><name="Headhunter1",password="xxxxx",domain="financial"> |

| | |
|---|---|
| | <name="registry2",password="yyyyy", domain="financial"><br>The DSM authenticates the principals and returns a secret key and multicast address to a valid principal (one to many interaction).<br>Example response:<br><secretkey = keyforFinancialDomain, multicast_address="224.2.2.2"> |
| **2** | This indicates the interactions between Service Exporter Components and active registries.<br>The service exporter components register with their respective registries (many to one interaction).<br>Example: <id="intrepid.cs.iupui.edu/AccountServer"><br>These registries in turn query these components for their UniFrame Specification (one to many interaction).<br>Example: <introspect property = "UniFrameSpecURL"><br>The components respond with the URL at which the specification is located (many to one interaction).<br>Example: <url="C:\Account System\AccountServerSpec.xml"> |
| **3** | This indicates the interactions between Headhunters and Active Registries.<br>Headhunters periodically multicast their presence to multicast group addresses (one to many interaction).<br>Example message: <Headhunterlocation=phoenix.cs.iupui.edu/Headhunter1><br>Active Registries, which are listening for the multicast messages from the Headhunters at this group address, respond to Headhunter's multicast messages by passing their contact information to the Headhunter (many to many interaction).<br>Example: <registrylocation=magellan.cs.iupui.edu/registry2><br>Headhunters query the active registries, which respond to their announcements, for the UniFrame specification information of all the components registered with them (one to many interaction).<br>The active registries respond by passing to the Headhunter the list of components registered with them and the detailed UniFrame specification of these components (many to many interaction). |

| | |
|---|---|
| **4** | This indicates the interactions between a Headhunter and a Meta-Repository. Headhunters store the component information obtained from the active registries onto the Meta-Repository (one to one interaction).<br>Headhunters query the Meta-Repository to retrieve component information (one to one interaction).<br>Example:<br><SQL statement = "SELECT * FROM *componentTable* A, *functionTable* B WHERE (A.ID = B.ID) AND ((*description* LIKE %account%) OR (*description* LIKE %system%)) AND (*end2endDelay* < 50) AND (*availability* > 50)"><br>Meta-Repository returns search results to Headhunter (one to one interaction). |
| **5** | This indicates the interactions between the QM and Component Assemblers/System Integrators.<br>System Integrators contact the QM and specify the functional and non-functional search criteria. The System Integrators can optionally specify the domain for their query or allow the natural language processor of the QM to determine the domain of the query. (many to one interaction).<br>Example: The natural language-like system integrator query is as follows:<br><NL Query=*"Create an account management system that has end-to-end delay < 50 ms and availability> 50% with preference maximum availability."*>.<br>The QM returns the search results to the clients (one to many interaction).<br>Figure 3.6 depicts an example of the search results returned to the system integrator.<br><br>**1. JavaAccountServer**<br>availability = 95%, End-toEnd Delay < 10 ms, Java-RMI Technology<br>**2. JavaAccountClient**<br>availability = 90%, End-toEnd Delay < 20 ms, Java Applet Technology<br>**3. CorbaAccountServer**<br>availability = 90%, End-to-End Delay < 15ms, CORBA Technology<br>**4. CorbaAccountClient**<br>availability = 90%, End-toEnd Delay < 20 ms, CORBA Technology<br>Figure 3.6. Example of Search Results. |

| | |
|---|---|
| **6** | This indicates the interaction between the QM and DSM.<br>QM contacts DSM for contact information of registered Headhunters belonging to the domain of client query (one to one interaction).<br>DSM responds with the list of registered Headhunters (one to one interaction).<br>Example:<br>< phoenix.cs.iupui.edu/Headhunter1, magellan.cs.iupui.edu/Headhunter2> |
| **7** | This indicates the interactions between the QM and Headhunters.<br>The QM propagates the System Integrator's query to all registered Headhunters, which fall in the domain of the System Integrator's search request (one to many interaction).<br>The Headhunters respond to the QM query with search results matching the criteria (many to one interaction). |
| **8** | This indicates the interactions between adapter components and AM.<br>Adapter components register with the AM, which is running at a well-known location (many to one interaction). |
| **9** | This shows the interactions between the clients and the AM.<br>Clients contact the AM at the well-known location at which it is running with requests for specific adapter components (many to one interaction).<br>The AM checks against its repository for matches and returns the results to the clients (one to many interaction). |
| **10** | This shows the interactions between QM and LM.<br>The QM propagates the query to the LM (one to one interaction).<br>LM returns search results to QM (one to one interaction). |
| **11** | This shows the interactions between the LM of one ICB and target LMs of other ICBs with which this LM is registered.<br>The LM propagates the search query issued by the QM to all the target LMs (one to many interaction).<br>The source LM receives the result responses from these target LMs (many to one interaction). |

Table 3.2. Interactions between URDS components

## 3.6 The UniFrame Approach (UA)

Figure 3.7 depicts the UniFrame Approach for the development of software solutions for DCS. This approach as proposed in [RAJ01] has two levels:

- Component Level - in this level, different components are created by developers, tested and verified from the point of view of QoS, and then deployed on the network
- System Level -this level concentrates on assembling a collection of components, each with a specific functionality and QoS, and semi-automatically generates the software solution for the particular DCS under consideration.

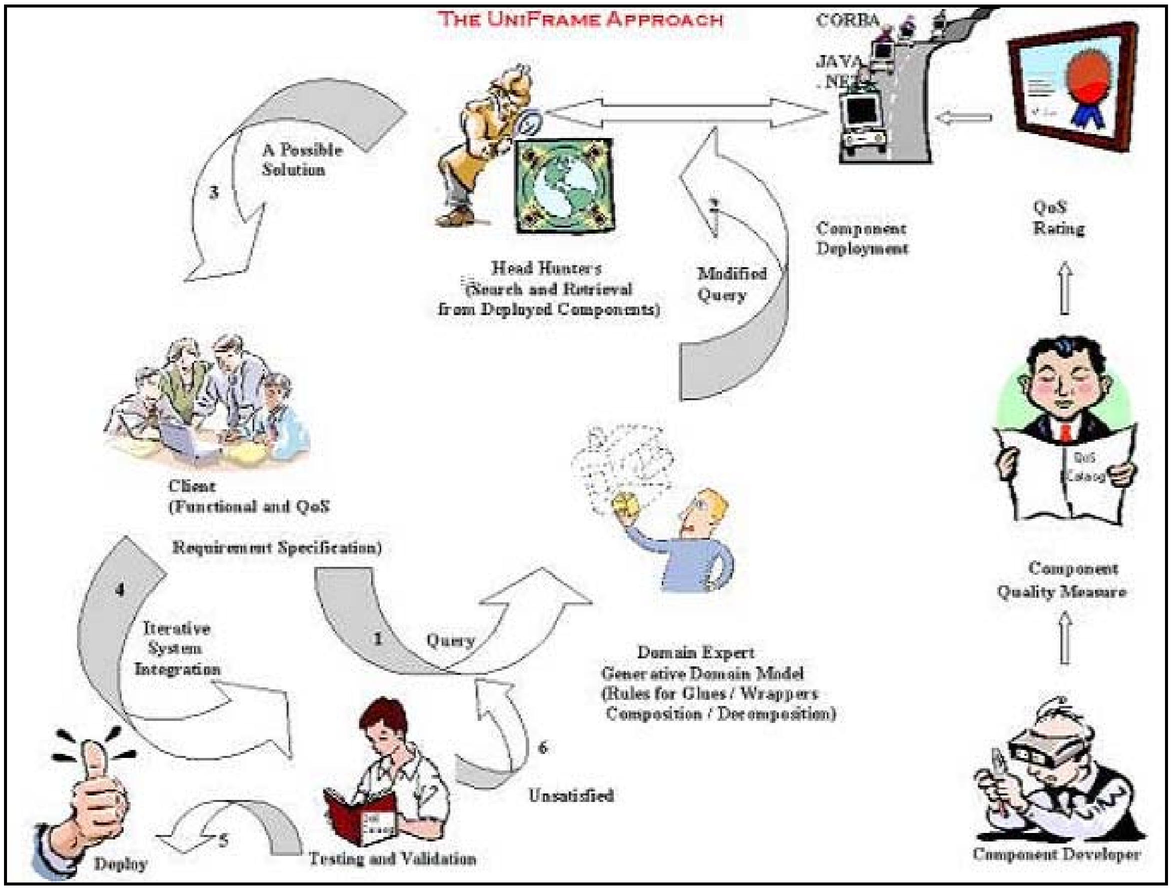

THE UNIFRAME APPROACH
CORBA
JAVA
.NET
A Possible Solution
3
Head Hunters (Search and Retrieval from Deployed Components)
2
Modified Query
Component Deployment
QoS Rating
Client (Functional and QoS Requirement Specification)
Component Quality Measure
4
Iterative System Integration
1
Query
Domain Expert Generative Domain Model (Rules for Glues / Wrappers Composition / Decomposition)
6
Unsatisfied
5
Deploy
Testing and Validation
Component Developer

Figure 3.7. UniFrame Approach

The two levels of the UA and associated processes as described in [RAJ01] are provided below.

## 3.6.1 Component Development and Deployment Process

The component development and deployment process (illustrated in Figure 3.8) starts with a UniFrame specification of a component from a particular domain (see Figure 3.2). This specification is in a natural language-like format and indicates the computational, cooperative, and auxiliary aspects and QoS metrics of the component. This informal specification is then refined into a XML-based specification (see Figure 3.3). The refinement is based upon the theory of Two-Level Grammar (TLG) natural language specifications [BAR00, VAN65], and is achieved by the use of conventional natural language processing techniques (e.g., see [JUR00]) and a domain knowledge base. The refinement process also includes the generation of interfaces, which may then be integrated into an implementation using glue and wrapper code. The interface incorporates all the aspects of the component, as required by the UniFrame specification. The developer provides the necessary implementation for the computational, behavioral, and QoS methods. This process is followed by the QoS validation. If the results are

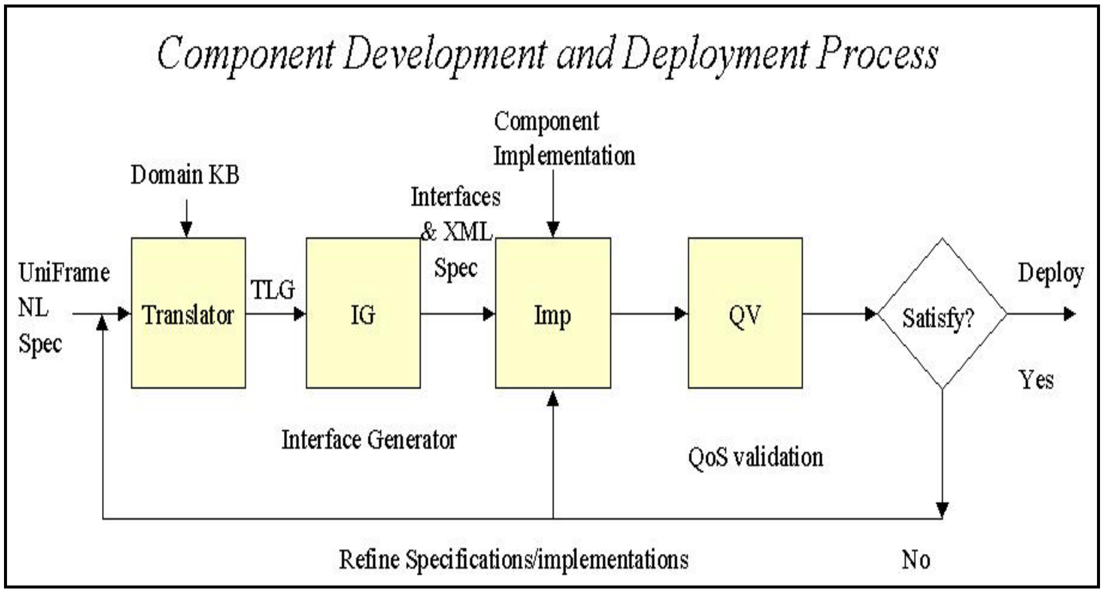


Figure 3.8. Component Development and Deployment Process

satisfactory (as required by the QoS criteria) then the component is deployed on the network and eventually, it is discovered by one or more Headhunters. If the QoS constraints are not met then either the developer refines the UniFrame specification or the implementation and the cycle repeats.

### 3.6.2 Automated System Generation and Evaluation based on QoS

In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available, then the task is to assemble them into a solution. The proposed framework takes a pragmatic approach, based on Generative Programming [CZA00], to component-based programming. It is assumed that the generation environment will be built around a generative domain specific model (GDM) supporting component-based assembly. The distinctive features of the proposed approach are as follows:

The developer of the desired distributed system presents to this process a system query, in a structured form of natural language that describes the required characteristics of the distributed system. The natural language processor (NLP) processes the query. It does this aided by the domain knowledge (such as key concepts from the domain) and a knowledge-base containing the UniFrame description of the components for that domain. From this query a set of search parameters is generated which guides Headhunter agents for a component search in the distributed environment.

The framework, with the help of the infrastructure described in Section 3.5, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. From these, the developer, or a program acting as a proxy of the developer, selects some components. After the components are fetched, the system is assembled according to the generation rules embedded in the generative domain model. These components along with the appropriate adapters (if needed) form a software implementation of the distributed system.

Next this implementation is tested using event traces and the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. Once a satisfactory implementation is found, it is ready for deployment.

The process for automated system generation and evaluation is illustrated in Figure 3.9.

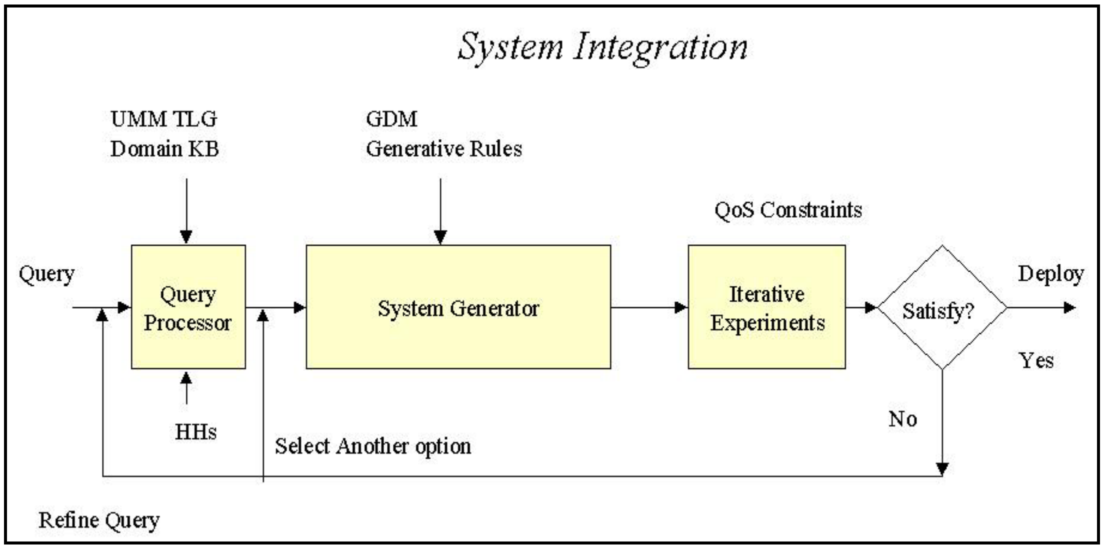


Figure 3.9. Automated System Generation and Evaluation

This chapter provided an overview of the UniFrame Approach to developing a software solution for a DCS. The next chapter elaborates on the URDS architecture by providing a detailed high-level description of the URDS components.

# 4. URDS ARCHITECTURE

Chapter 3 provided an overview to the UniFrame Approach and the URDS architecture. The present chapter describes the URDS architecture in detail, focusing on the high-level design, the algorithms, and the interactions of the components that comprise the URDS architecture. The descriptions in this chapter are at a conceptual level and are not tied to the software that may implement the architecture. The implementation specific details will be covered in Chapter 5.

Figure 4.1 below illustrates the components of the URDS architecture outlined in Chapter 3.

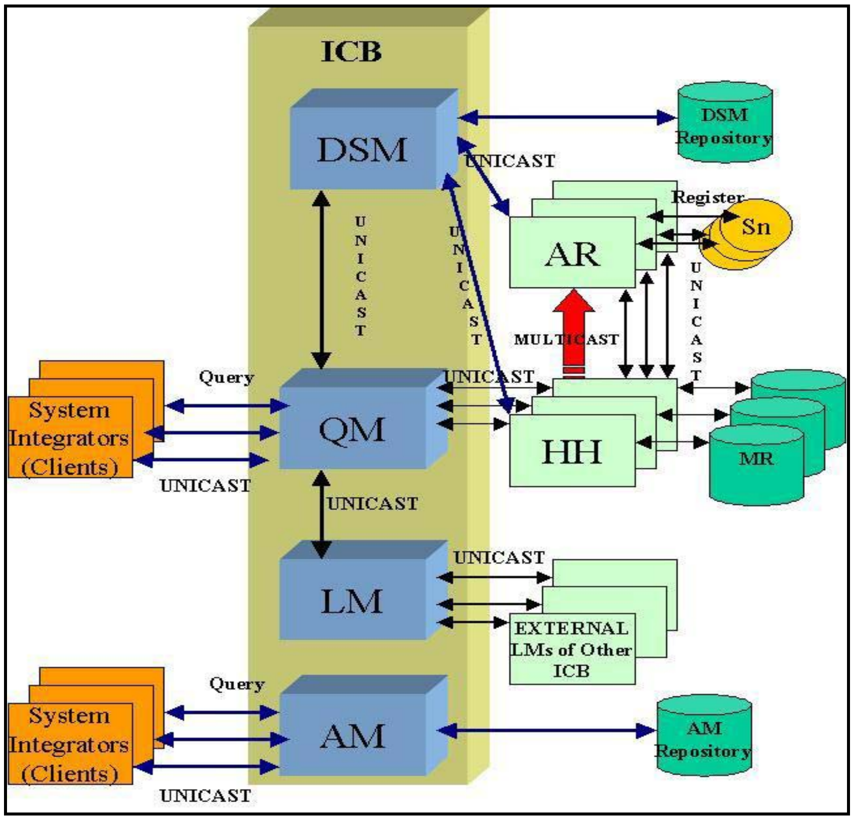


Figure 4.1. Components of the URDS Architecture

The overview of the URDS architecture presented in Chapter 3 highlighted the functions of its components namely: i) Internet Component Broker (ICB) which is a collection of the following services - Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM) ii) Headhunters (HHs), iii) Meta-Repositories (MR), and iv) Active-Registries (AR). The URDS architecture is designed to discover heterogeneous services (S1..Sn). The clients/users of the URDS system are the Component Assemblers, System Developers or System Integrators.

The following subsections provide the high-level design details and algorithms for each of the components in the URDS architecture.

## 4.1 Internet Component Broker

The ICB acts as an all-pervasive component broker in the interconnected environment providing a platform for the discovery and seamless integration of disparate components. The ICB is not a single component but is a collection of services comprising of the Query Manager (QM), the Domain Security Manager (DSM), Adapter Manager (AM), and the Link Manager (LM). It is envisioned that there will be a fixed number of ICBs deployed at well-known locations hosted by corporations or organizations supporting the UniFrame initiative. The functionality of the ICB is similar to that of an Object Request Broker. However, the ICB has certain key features that are unique. It provides component mappings and component model adapters. The ICB, in conjunction with Headhunters, provides the infrastructure necessary for scalable, reliable, and secure discovery and registry services using the interconnected infrastructure.

The chief functionalities provided by the ICB are:

- Authenticate the principals (Headhunters and Active Registries) in the system and enforce access control over the multicast address resources for a domain with the help of the Domain Security Manager (DSM).
- Attempt at matchmaking between service producers and consumers with the help of the Headhunters and Query Manager. ICBs may cooperate with each other in order to increase the search space for matchmaking. The cooperation techniques of ICBs are facilitated through the Link Manager (LM).
- Act as a mediator between two components adhering to different component models. The mediation capabilities of the ICB are facilitated through the Adapter Manager (AM).

### 4.1.1 Algorithm for ICB Initialization

In the algorithm the ICB initializes all the services i.e., the DSM, QM, LM and AM by calling their initialization routines.

```
ICB_INITIALIZATION
    CALL DSM_INITIALIZATION (see Algorithm 4.2.4.1)
    CALL QM_INITIALIZATION (see Algorithm 4.3.3.1)
    CALL LM_INITIALIZATION (see Algorithm 4.4.1.1)
    CALL AM_INITIALIZATION (see Algorithm 4.5.1.1)
END_ICB_INITIALIZATION
```

The following subsections provide a description of the services that comprise the ICB and their contribution to providing the functionality of the ICB.

## 4.2 Domain Security Manager (DSM)

The URDS discovery protocol is based on periodic multicast announcements. Multicasting [ERI94] is a scalable solution for the group communication. However, there are various security issues that have to be addressed in a multicast communication, which do not arise in an unicast communication [BAL95]. Security threats that need to be addressed are:

- *Eavesdropping*: Since multicast addresses are not private, any interested host can gain access to the multicast data by just becoming a member of that group. These kinds of security attacks, where adversaries try to gain access to the data without really disrupting the secure multicast protocol, are called passive attacks [STA95]. Potential adversaries may also use the data collected by means of eavesdropping for cryptanalysis purposes or replay the data at a later time.
- *Uncontrolled Group Access*: Since access to the group is not controlled in multicast communication, any host can send data to a multicast group which can cause congestion and also possibilities of a *Denial Of Service* attack against the group.
- *Masquerading:* An adversary may also pose as another host that is a member of the group. This enables the adversary to send data, receive data and or acquire access to the secret-keys by posing as a legitimate member of the group. Masquerading is a form of active attack.

[DON99] has identified the following components; a) *Group Membership Control, b) Secure Communication*, and *c) Key Distribution* as being critical to effectively combating the security threats faced in multicast communication. The following subsections provide a brief overview of these security mechanisms and explain how they are incorporated in the URDS architecture with the help of the DSM.

### 4.2.1 Group Membership Control (GMC)

The GMC allows only authorized hosts to join a multicast group, guarding against otherwise unilateral subscriptions by arbitrary hosts. Group membership control can be exercised in two ways:

- *Access Control Lists:* Access Control Lists allow a sender or an authorized third party to maintain an inclusion or an exclusion list of hosts on the Internet corresponding to a multicast group. Each time a host requests to join the multicast group, the sender or the third party checks with the access control list to determine whether the host is authorized to join the group. [ACC97] defines an ACL as a "data structure that guards access to resources". An ACL data structure is comprised of multiple entries, each of which contains a set of permissions associated with a particular principal. A principal represents an entity such as an individual user or a group. Each entry can be specified as being either positive (i.e., permissions are granted to the associated principal) or negative (i.e., permissions are denied). An ACL is independent of the authentication scheme used to verify the validity of the principal, the encryption scheme used to transmit the data across the network and the resource that it guards. The ACL is consulted after the authentication phase. After the principal is verified to be an authenticated user in the system, the principal may access resources. For each such resource, the principal may or may not be granted an access depending on the permissions that are granted to the principal in the ACL that guards the resource. The ACL can be consulted to find the list of permissions a particular principal has or to find out whether or not a principal is granted a particular permission [ACC97]. The details of how the ACL is created is presented in the algorithm for creating ACL entries in subsection 4.2.4.2.

- *Use of Capability Certificates* [DON99]: Capability Certificates are issued by a designated third party to receivers. Capability Certificates authenticate the receivers and authorize them to participate in a multicast group. The receivers present the capability certificates to the sender or an authorized third party to gain access to the group.

The Group Membership Control in the URDS is enforced through the use of userid/password authentication and Access Control Lists. The resources being guarded are the multicast addresses allocated to a particular *domain*. The Domain Security Manager (DSM) serves as an authorized third party, which maintains an inclusion list of Principals (Headhunters or registries) corresponding to a domain. DSM has an associated repository (database) of valid principals, passwords, multicast address resources and domains. Every Headhunter or Active Registry is associated with a domain. The Active Registries associated with a domain have components registered with them, which belong to that domain. The Headhunter in turn detects Registries, which belong to the same domain as itself, and hence, the service components detected by the Headhunter will belong to a particular domain. The Principal (authenticated user), is allowed access only to the multicast address mapped to the domain with which it is associated. A Principal that wishes to participate in the discovery process contacts the DSM with its credentials (id, password, domain). The DSM authenticates the principal and checks its authorizations against the domain ACL. The DSM returns a secret-key and a multicast address mapped to the corresponding domain to a valid principal. The DSM_Repository is created and populated by an administrator. The mappings between multicast addresses and domains, and users and domains are established by the organization providing the DSM service.

### 4.2.2 Secure Communication

A Secure communication can be ensured by the transmission of encrypted data. There are several cryptographic mechanisms such as secret-key [STA95] and public-key [STA95] mechanisms.

In the URDS, the security of transmitted data is ensured through the process of Secret-Key Encryption. A secret-key is a symmetric key wherein the sender and receiver use the same key for purposes of encryption and decryption. The DSM is responsible for generating a unique secret-key for each domain.

### 4.2.3 Key Distribution

A Key distribution scheme should ensure that the security key is not compromised and the communication does not yield to active security attacks such as masquerading. Many schemes exist for securing key distribution such as Centralized Flat Key Management Scheme [BLU97, CAR98, GON94, HAR97], Hierarchical Key Management Scheme [BAL96, CAR98, MIT97, WAL97], and Distributed Flat Key Management Scheme [CAR98].

The URDS follows a Centralized Flat Key Management scheme wherein the DSM serves as the central authority, which generates and distributes the secret-key to authenticated principals. An overview of the DSM Security Enforcement Protocol is provided below:

- DSM acts as an authorized third party centralized controller for the secret-key distribution and the multicast address allocation to Headhunters and Registries.
- Secret-Key and Address allocation is domain specific.
- Headhunter/Registries communicate to DSM their authentication credentials (id, password, domain).
- DSM verifies authentication credentials and checks permissions against the ACL to verify the user access to that domain.
- DSM returns to the authorized users a multicast address picked at random from the list of addresses allocated to that domain and a secret-key generated for that domain.

- In case the principal is a Headhunter the DSM registers the contact information of the Headhunter with itself. The QM uses this information for the query propagation.

## 4.2.4 Algorithms for DSM Functions

Data structures used:

| | |
|---|---|
| Hash table[1] *userDomainTable* | Mapping between principal names and their corresponding domains. The principal name serves as the key for this mapping. |
| Hash table *domainTable* | Mapping between multicast addresses and corresponding domain names. The multicast address serves as the key for this mapping. |
| Hash table *HHAddressAllocTable* | Mapping between multicast addresses allocated to Headhunters and their corresponding domains. The multicast address serves as the key. |
| Hash table *registeredHHTable* | Mapping between registered Headhunter locations and their domain. The registered Headhunters are the authorized principals who have received a multicast address and secret-key from the DSM. |
| Hash table *keyTable* | Mapping between domains and corresponding Secret-Keys. The domain names serve as the key. |
| ACL[2] *domainACL* | ACL of usernames and their associated permissions (i.e. domains). |

Table 4.1. Data Structure for DSM functions

[1] The algorithms in this section and succeeding sections use the hash table ADT to represent key-value pair mappings. These hash tables are maintained as memory resident tables.

[2] The ACL is maintained in memory after creation.

### 4.2.4.1 Algorithm for DSM Initialization

The DSM configuration process comprises of setting up the DSM_Repository with the information about the domains, associated multicast addresses, authorized users and their passwords. The information is stored in the DSM_Repository as a collection of tables (refer sub section 5.3.4.1.1.3 in Chapter 5 for implementation specific details on these tables). The DSM configuration is performed by an administrator. The DSM upon initialization creates an ACL of usernames and their associated permissions and generates secret-keys for each of the domains. It then actives the authentication service, which can respond to remote calls by the Headhunters and active registries. New entries to the DSM_Repository are added on an as-needed basis and the in-memory ACL is updated accordingly.

```
DSM_INITIALIZATION
        CREATE DSM_REPOSITORY
        CALL DSM_CREATE_ACL_ENTRIES
        CALL DSM_GENERATE_SECRET_KEYS
        ACTIVATE DSM_AUTHENTICATION_SERVICE
END_DSM_INITIALIZATION
```

### 4.2.4.2 Algorithm for Creating Entries in the ACL

This algorithm outlines the process for creating principals and permissions using usernames and domain names. It also describes the process for creating ACL entries using these principal-permission pairs and adding these entries to the ACL.

```
DSM_CREATE_ACL_ENTRIES
    LOAD userDomainTable from DSM_Repository
    userNameList = GET enumeration of user names from userdomainTable
    WHILE userNameList HAS MORE ELEMENTS
        username = GET NEXT ELEMENT from userNameList
        domainName = GET value in userdomainTable for username key.
        Create a new PRINCIPAL with the username
        Create a new ACLENTRY with the new PRINCIPAL
        Create a new PERMISSION with the domainName
        Add PERMISSION to ACLENTRY
        Add ACLENTRY to domainACL
    ENDWHILE
END_DSM_CREATE_ACL_ENTRIES
```

#### 4.2.4.3 Algorithm for Generating Secret-Keys

This algorithm outlines the process for generating secret-keys using any standard secret-key-generating algorithm such as DES, Triple DES, Blowfish, etc. A secret-key is generated for each domain and stored in a hash table ADT of domain names and secret-key pairs.

```
DSM_GENERATE_SECRET_KEYS
    domains = GET enumeration of domain names from domainTable
    //Create a key generator component capable of generating secret-keys that
    // corresponds to a given key generating algorithm.
    keyGenerator =
    CREATE a new KEYGENERATOR with keyGeneratingAlgorithm
    //Generate a secret-key for each domain.
```

```
        WHILE domains HAS MORE ELEMENTS
                domainName = GET NEXT ELEMENT from domains
                secretKey = GENERATEKEY with keyGenerator
                PUT < domainName, secretKey > in keyTable
        ENDWHILE
END_DSM_GENERATE_SECRET_KEYS
```

4.2.4.4 Algorithm for Authentication, and Secret-Key and Multicast Address Distribution

This algorithm outlines the process for authenticating a principal by verifying their authentication credentials against the DSM_Repository. The authorization of user permissions is done by checking against the ACL. A multicast address is picked at random from the list of multicast addresses allocated to that domain. An authenticated packet comprising of the multicast address and secret-key for the domain is returned to the user.

```
DSM_AUTHENTICATION_SERVICE
        IN: userType, userName, password, contactLocation, domain
        OUT: authenticatedPacket containing multicastAddress and secretKey
        WHILE TRUE
                IF contacted by user
                        isUserAuthenticated =
                                VALIDATE userType, userName, password against
                                DSM_Repository.
                        IF isUserAuthenticated EQUALS TRUE
                                CREATE a new PRINCIPAL for the username
                                CREATE a new PERMISSION for the domain
                                isUserAuthorized = CHECK PRINCIPAL's PERMISSION
                                                   in domainACL
```

```
IF isUserAuthorized EQUALS TRUE
    /* If a multicast address for a domain is requested
    by a Registry and there are addresses allocated to
    Headhunters for this domain  then get the list of
    domainAddresses from HHAddressAllocTable
    corresponding to domainName and select an address
    at random from this list. */
    IF (userType EQUALS "Registry") AND
    (HHAddressAllocTable CONTAINS VALUE
    domainName )
        mcastaddressList =
        GET the list of domainAddresses from
        HHAddressAllocTable corresponding to
         domainName.
        domainAddress =
        SELECT RANDOM address from
                 mcastaddressList
    ELSE
        /* If a HeadHunter/Registry requests
        multicast address for a domain pick an
        address at random from the domainTable–
        i.e. from the list of addresses allocated to
        this domain. */
        mcastaddressList =
        GET list of domainAddresses from
        domainTable corresponding to
        domainName.
        domainAddress =
        SELECT RANDOM address from
        mcastaddressList
```

```
                        IF (userType EQUALS "Headhunter")
                        /* Maintain a list of Headhunter contact
                        locations */
                        PUT<contactLocation, domainName> in
                        registeredHHTable
                         /* Maintain a list of addresses-domain
                        mappings allocated to Headhunters. */
                        PUT <domainAddress, domainName> in
                         HHAddressAllocTable
                         ENDIF
                    ENDIF
                    secretKey =
                    GET secretkey from keyTable corresponding to
                                        domainName
                    CREATE authenticatedPacket and Send
                    Response to user
                ENDIF //authorized user
            ENDIF //authenticated user
        ENDIF //contacted by user
    ENDWHILE
END_DSM_AUTHENTICATION_SERVICE
```

4.2.4.5 Algorithm for Withdrawing  Headhunters from DSM

This algorithm outlines the process for withdrawing a Headhunter from the DSM.

```
DSM_WITHDRAW
    IN: HeadhunterLocation
    REMOVE key-value pair corresponding to HeadhunterLocation from
```

```
        RegisteredHHTable
END_DSM_WITHDRAW
```

## 4.3 Query Manager (QM)

The QM is responsible for finding services matching the client’s query requests. The QM uses a parser [LEE02] to translate a service consumer’s natural language-like query into an XML-based query. The QM parses the XML based query to generate a structured query language statement and dispatches this query to the ‘appropriate’ Headhunters (determined by the domain of the query). Requests for service components belonging to a specific domain will be dispatched to Headhunters belonging to that domain and the Headhunters return the list of matching service providers to the QM. The QM obtains the list of registered Headhunters from the DSM. The QM in conjunction with the LM is also responsible for propagating the queries to other linked ICBs.

The QM is configured by an administrator with a *Query Propagation Policy.* The QM propagates a query to the LM based on the *Query Propagation Policy* specified. The *Query Propagation Policy* can be specified as:

- *If No Local*: Propagate a query to the LM only if there are no local search results that satisfy the query
- *Always*: Always propagate a query to the LM

### 4.3.1 Result Selection Process

The total search space for a service selection may be very large; including services from all Headhunters associated with a particular Internet Component Broker (ICB) and all linked ICBs. The QM uses policies (explained in subsection 4.3.1.1) to identify a set S1 of services to return. The QoS type and constraint (explained in subsection 4.3.1.2) are applied to S1 to produce the set S2 that satisfies the service type and constraint. Then this can be is ordered using the preferences specified in the query (explained in subsection 4.3.1.3), before returning the services to the service requesters.

#### 4.3.1.1 Policies

Policies provide information to affect QM behavior at run time. Policies are represented as name-value pairs. Policies can be grouped into two categories: a) policies that scope the extent of a search, and b) policies that determine the functionality applied to an operation. The examples for *search scoping policies* are i) upper bound of offers to be searched, ii) upper bound of offers to be returned, and iii) upper bound of matched offers to be ordered, etc. Examples for *functionality scoping policies* are minimum number of QoS types desired to be matched.

#### 4.3.1.2 Constraints

Service Requesters use QoS types and constraints to select the set of service offers in which they have an interest. Constraints are described by the 3-tuple <*QoS-type-name, Operators, Literals*> where *QoS-type-name* specifies the QoS metric on which the constraint is being applied, *Operators* are comparison, boolean connective, substring, arithmetic operators, etc., and *Literals* are the numeric/string/boolean values corresponding to the QoS types. Example: <availability, >, 50>

#### 4.3.1.3 Preferences

Preferences are applied to the set of services matched by application of the QoS type, constraint expression, and various policies. The application of the preferences can determine the order used to return matched services to the service requesters. Preferences are associated with the 2-tuple <*Sort Order, QOS-type-name* >. The sort order can be specified as *max* (matched offers are returned in descending order of QoS-type values), *min* (matched offers are returned in ascending order of QoS-type values), and *first* (matched offers are returned in the order they are discovered).

### 4.3.2 Query Handling Process

The QM Query Request Handling Protocol consists of the following steps:

- Parse a component assembler's/system integrator's natural language-like query and extract the keywords and phrases pertaining to various attributes of the components UniFrame specification.
- Extract the consumer-specified constraints, preferences and policies to be applied to the various attributes.
- Compose the extracted information into an XML-based query.
- Translate the XML-based query to a structured query language statement.
- Dispatch this structured query to all the Headhunters associated with the domain on which the search is being performed and also forward the query based on the *Query Propagation Policy* to the Link Manager, which will propagate the query to other ICBs.
- The Headhunters will query associated Meta-Repositories and return a list (possibly non-empty) of components matching the search criteria to the QM, which is returned to the system integrator making the query.
- QM will wait for a specified time period for results to be returned from the Headhunters/other ICBs before timing out. A default timeout period for a session is configured into the QM by an administrator. A session constitutes each new query operation serviced.
- The system integrator has the option to specify search-scoping policies to affect the time spent on the search process.

### 4.3.3 Algorithms for QM Functions

Data structures used:

| | |
|---|---|
| Entity[3] *queryEntity* | An entity, which holds all the attributes corresponding to the client's query request with accessors for accessing the attributes. The attributes of the *queryEntity* include the computational, cooperational, and auxiliary attributes, QoS Metrics, QM Policies, Constraints, and Preferences. |
| Hash table *resultTable* | Mapping between Component IDs and components holding the detailed UniFrame Specification. |

Table 4.2. Data Structure for QM functions

[3] An Entity data structure is a conceptual representation for an object and holds all the attributes that describe the object with accessors for accessing these attributes.

#### 4.3.3.1 Algorithm for QM Initialization

The QM initialization activates the client request handler. This process receives requests from clients and responds with results. It also activates the query propagation service, which can be called by QM or LM.

```
QM_INITIALIZATION
    ACTIVATE QM_CLIENT_REQUEST_HANDLER
    ACTIVATE QM_PROPAGATE_QUERY
END_QM_INITIALIZATION
```

4.3.3.2 Algorithm for Handling Query Requests from Clients

This algorithm outlines the process for servicing requests from Clients. Clients contact the QM with a natural language-like query. This is processed into an XML query, which is parsed to construct an entity that embodies all the attributes of the query. The *queryEntity* is propagated to Headhunters, which are associated with the domain of the query and also to the LM based on the results obtained from the Headhunters and the *QueryPropagationPolicy* set on the QM. The final results returned from the Headhunters and LMs are ordered as per the client's preferences and returned to the client. Each client request is handled as a separate session. The session timeout period is configured in the QM.

```
QM_CLIENT_REQUEST_HANDLER
    IN: naturalLanguageQuery
    OUT: resultTable
    //Get handle to DSM and LM by contacting them at the well-known locations.
    dsm = LOOKUP dsmLocation
    lm = LOOKUP lmLocation
    WHILE TRUE
        IF contacted by client with naturalLanguageQuery Request
            /* The natural language query is parsed and keywords, constraints,
            preferences, and policies extracted and domain knowledge base is
            applied to the extracted information and written to a XML File */
            Parse naturalLanguageQuery and generate XML File.
            /* The XML query is parsed and the values populated into an
            entity which contains the logic to construct the structured query
            language statement. */
            xmlQueryDocument =
            Parse XML file and load XML document into memory.
            /* The QM_PROCESS_XML_QUERY function populates
```

```
the queryEntity instantiated here. */
CREATE a new queryEntity
CALL QM_PROCESS_XML_QUERY with xmlQueryDocument
//Propagate the query to all Headhunters in the domain of query
resultTable =
CALL QM_PROPAGATE_QUERY with queryEntity
/* Check if the search scoping policies have been met
example if upper bound of offers to be returned
has been met and if so return results to client. */
searchScopePolicy = GET search scoping policy from queryEntity
IF (searchScopePolicy is specified) AND
   (searchScopePolicy is satisfied)
        resultTable = CALL QM_ORDER_RESULTS
        with queryEntity, resultTable
        Send response to client with resultTable
ENDIF
/* If further propagation of query is required
Check the Query Propogation Policy on the QM.
 If QueryPropogationPolicy = "IfNoLocal" and
 there are no results matching the search criteria
then propagate query.
If the QueryPropogationPolicy = "Always"
 then propagate query irrespective of whether there are local
 search results. */
IF ((resultTable NOT NULL)) OR
( (QueryPropogationPolicy EQUALS "Always"))
        results = CALL LM_PROPAGATE_QUERY on lm
         with queryEntity
        ADD results to resultTable
ENDIF
```

```
                //Order the results as per client preferences
                resultTable =
                CALL QM_ORDER_RESULTS with queryEntity, resultTable
                Send response to client with resultTable
            ENDIF
        ENDWHILE
END_QM_CLIENT_REQUEST_HANDLER
```

### 4.3.3.3 Algorithm for Processing XML Query

This algorithm uses recursion to parse through the nodes of the XML tree, extract the node values and store these values in the *queryEntity*. The algorithm starts parsing at the root node element. It extracts the node name and checks if the node name matches any attribute in *queryEntity* and populates it with the value in this node. It then finds all the children of that node and repeats the process through recursion.

```
QM_PROCESS_XML_QUERY
    IN: nodeElement
    nodeName = GET NODE NAME from nodeElement
    FOR each attribute in queryEntity
        IF nodeName EQUALS attribute
            nodeValue = GET NODE VALUE from nodeElement
            SET attribute value in queryEntity to nodeValue
        ENDIF
    ENDFOR
    // Get the list of children for this Node.
    NODELIST childrenList = GET CHILDNODES for nodeElement
    IF childrenList NOT NULL
        // For every child node in the list
        FOR i = 0 to LENGTH of childrenList
            childNode = childrenList[i]
            /* Recurse through the function PROCESS_XML_QUERY
            passing  it the childNode as reference.*/
            CALL QM_PROCESS_XML_QUERY with childNode
        ENDFOR
    ENDIF
END_QM_PROCESS_XML_QUERY
```

4.3.3.4 Algorithm for Propagating Query to Headhunters

This algorithm outlines the process for popagating a query request to the Headhunters. The domain of a client's query and the search scoping policies are retrieved from the *queryEntity*. The DSM is contacted to obtain a list of registered Headhunter locations matching this *domain*. The query is then propagated to each Headhunter and the search scoping policies are checked after each propagation to verify whether the policies have been satisfied. If the policies are satisfied then further propagation is terminated and the results are ordered as per client preferences and returned.

```
QM_PROPAGATE_QUERY
    IN: queryEntity
    OUT: resultTable
    WHILE TRUE
        IF contacted by this QM or LM
            //The queryEntity stores the details of the query such as the domain
            // and other attributes of the query.
            domain  = GET domainName of query from queryEntity
            /* Get location list of registered Headhunters for the domain of the
            query. */
```

```
                    hhLocationList = GET Headhunter locations list from dsm with
                                    domain
                    //Propagate the query to each registered Headhunter in the domain.
                    FOR i=0 to LENGTH of hhLocationList
                        //Get the location of the Headhunter
                        hhlocation = hhLocationList[i]
                        //Get a handle to the Headhunter
                        Headhunter = LOOKUP hhlocation
                        //Propagate query to each Headhunter.
                        results = CALL  HH_EXECUTE_QUERY
                                on Headhunter with queryEntity
                        ADD results to resultTable
                    ENDFOR
                    RETURN resultTable
                ENDIF
        ENDWHILE
END_QM_PROPAGATE_QUERY
```

4.3.3.5 Algorithm for Ordering Search Results

This algorithm outlines the process for retrieving client preferences (sort order ascending or descending) and the QoS type on which the ordering is to be performed. The results are then ordered accordingly and returned.

```
QM_ORDER_RESULTS
        IN: queryEntity, resultTable
        OUT: resultTable
        orderPreference = GET ordering preference from queryEntity
        qosTypeToOrder = GET the QoS-Type-Name to order by from queryEntity
```

```
    IF orderPreference NOT NULL // If an order preference was specified
        resultTable= SORT elements of resultTable on qosTypeToOrder
        RETURN resultTable
    ELSE
        RETURN resultTable to client
    ENDIF
END_QM_ORDER_RESULTS
```

### 4.3.3.6 Algorithm for Generating Structured Query Language (SQL) Statement

This algorithm forms a part of the operations performed by *queryEntity*. This algorithm outlines the process for generating a SQL statement based on the attributes extracted from the client's query. These attributes are captured in the *queryEntity.* The attribute names, values and constraints stored in the query entity are built into a SQL statement.

```
QE_GENERATE_SQL_QUERY
    IN: queryEntity
    OUT: sqlQuery
    attributeList = GET all attributes in queryEntity
    FOR i=0 to LENGTH of attributeList
        attribute = attributeList[i]
        IF attribute is selected as search parameter
            attributeValue = GET value of this attribute from queryEntity
            attributeConstraint = GET constraint value from queryEntity
            CONCATENATE to BUILD SQL query the
            attribute, attributeConstraint, and attributeValue
        ENDIF
    ENDFOR
    RETURN sqlQuery
END_QE_GENERATE_SQL_QUERY
```

## 4.4 Link Manager (LM)

ICBs are linked to form a *Federation of Brokers* (refer Figure 3.5 in Chapter 3) in order to allow for an effective utilization of the *distributed offer space.* ICBs may propagate the search query issued by the system integrator to other ICBs to which they are linked. This linkage of ICBs makes the offer spaces of the linked ICBs implicitly available to the ICB's own clients.

The LM performs the functions of the ICB associated with establishing links and propagating the queries. *Links* represent paths for propagation of queries from a source ICB to a target ICB. Each link is unidirectional and corresponds to an edge in a directed graph, in which the vertices are LMs.

The Link Manager supports the following operations:

- *Query:* The query operation is responsible for propagating the query from the source LM to the list of Target LMs maintained with the Source LM.

- *Failure Detection:* This involves keeping track of LMs, which may no longer be active due to network failure, node failure, etc. Periodically ($TP_{unicast}$ ms – time period between successive failure detection cycles) the source LM contacts the list of target LMs with which it is configured, to re-establish links and to create a fresh target LM list. If the target LMs respond to the source LM, it means the Target LM is 'alive' to service requests. This target LM location is added to the list of target LMs to whom requests will be propagated in that cycle. If the target LMs do not respond then that target LM may have failed. During subsequent cycles if a previously failed target LM is now alive to service requests, then it is added to the list of LMs to receive queries for that cycle.

### 4.4.1 Algorithms for LM Functions

Data structures used:

| | |
|---|---|
| List *aliveTargetLMList* | List of Target LM locations, which are alive and can service query requests. |
| List *targetLMList* | List of locations of LM's with which this LM registers. This can be specified with the configuration information at LM initialization. |

Table 4.3. Data Structure for LM functions

#### 4.4.1.1 Algorithm for LM Initialization

The LM upon initialization establishes links with the list of target LMs passed to it as configuration information. It starts failure detection processes, which stay alive as an active processes and runs periodically in the background. It also activates the query propagation and link checking processes, which can be called from remote LMs.

```
LM_INITIALIZATION
    IN: targetLMList
    CALL LM_ESTABLISH_LINKS
    START LM_FAILURE_DETECTION
```

```
        ACTIVATE LM_PROPAGATE_QUERY
        ACTIVATE LM_CHECK_LINK
END_LM_INITIALIZATION
```

<u>4.4.1.2 Algorithm for Establishing Links with Target LMs</u>

This algorithm outlines the process followed by the LM for establishing links with Target LMs. The LM steps through the list of target LM locations it is configured with and contacts each of these target LMs. If the target LM responds, then this target LM location is added to the *aliveTargetLMList,* which maintains a list of all alive target LMs to whom the queries will be propagated. If the LM does not repond due to link or node failure then the failed target LM address is not stored.

```
LM_ESTABLISH_LINKS
        CREATE aliveTargetLMList
        //Contact the LMs specified in configuration.
        FOR i=0 to LENGTH targetLMList
                targetLMLocation = targetLMList[i]
                targetLM = LOOKUP targetLMLocation
                isAlive = CALL LM_CHECK_LINK on targetLM
                //If target LM is alive then add this target LM location
                //to the list.
                IF isAlive
                        //Maintain a list of LMs with whom registration was
                        // successful
                        ADD  targetLMLocation to aliveTargetLMList
                ENDIF
        ENDFOR
END_LM_ESTABLISH_LINKS
```

#### 4.4.1.3 Algorithm for Checking whether Links can be Established

The LM contacts target LMs to check if they are functioning. When contacted by a source LM the target LM returns a "true" response indicating it is alive to service requests.

```
LM_CHECK_LINK
    OUT: isAlive
    WHILE TRUE
        IF contacted by LM
            RETURN TRUE
        ENDIF
    ENDIF
END_LM_CHECK_LINK
```

#### 4.4.1.4 Algorithm for Propagating a Query by the LM

This algorithm outlines the process for receiving a query from the QM in the same ICB and propagating it to LMs in other ICBs or receiving a query from LMs in other ICBs and propagating query to the QM in the same ICB. The determination of where the query is coming from is made based on the value of the *requestID* attribute in *queryEntity*. The *requestID* is a *String* value, which is set to the value *"LinkManager"* if the query is received from an external LM or holds a default value of *"QueryManager"* when the *queryEntity* is instantiated by the QM for propagation. When the LM receives a query it inspects the value of the *requestID* attribute. If the *requestID* is set to *"QueryManager"* it means the query has originated from a QM in the same ICB and needs to be propagated to other LMs. If the *requestID* set to "LinkManager", it means the query is coming from another LM and needs to be propagated to the QM in the same ICB as the LM.

```
LM_PROPAGATE_QUERY
    IN: queryEntity
    OUT: resultTable
    WHILE TRUE
        IF contacted by QM in same ICB or LM in different ICB
            requestID = GET requestID from queryEntity
            //If the propagation request has come from LM in the different ICB
            IF (requestID EQUALS "LinkManager")
                //Propagate the request to QM located in same ICB as LM.
                resultTable = CALL QM_PROPAGATE_QUERY on qm
                        with queryEntity
                RETURN resultTable
            //If the propagation request has come from QM in the same
            //ICB as LM. The requestID would be set to "QueryManager".
            ELSE
                //This query will now be propagated to target LMs
                //hence the requestID parameter has to reflect this
                //that it is coming from a LM.
                SET requestID = "LinkManager" in queryEntity
                CREATE resultTable
                //Get list of all alive LMs to whom the query can be
                //propagated.
                FOR i = 0 to LENGTH of aliveTargetLMList
                    targetLMLocation = aliveTargetLMList[i]
                    targetLM = LOOKUP targetLMLocation
                    results =
                    CALL LM_PROPAGATE_QUERY on targetLM
                    with queryEntity
                    ADD results to resultTable
                ENDFOR
                RETURN resultTable
            ENDIF
        ENDIF
    ENDWHILE
END_LM_PROPAGATE_QUERY
```

<u>4.4.1.4 Algorithm for Failure Detection of Target LMs</u>

The LMs periodically check whether all the target LMs maintained in the target LM list are still alive by establishing contact with the target LMs.

```
LM_FAILURE_DETECTION
    WHILE TRUE
        CALL LM_ESTABLISH_LINKS
        // Periodically ($TP_{unicast}$ millisecs) refresh links so that
        //a list of 'alive' Target LMs is maintained. The optimal
        // $TP_{unicast}$ time is determined based on experimentation.
        SLEEP $TP_{unicast}$
    ENDWHILE
END_LM_FAILURE_DETECTION
```

## 4.5 Adapter Manager (AM)

The AM serves as a registry/lookup service for clients seeking adapter components. The adapter components register with the AM and while doing so they indicate their specialization (i.e., which heterogeneous component models they can bridge efficiently). System Integrators contact the AM to search for adapter components matching their needs. The AM utilizes adapter technology, each adapter component providing translation capabilities for specific component architectures. Thus, a computational aspect of the adapter component indicates the models for which it provides interoperability. The adapter components achieve interoperability using the principles of wrap and glue technology [LUQ01]. A reliable, and cost-effective development of wrap and glue is realized by the automatic generation of glue and wrappers based on component specifications. Wrapper software provides a common message-passing interface for components that frees developers from the error prone tasks of implementing interface and data conversion for individual components. The glue software schedules time-constrained actions and carries out the actual communication between components.

### 4.5.1 Algorithms for AM Functions

Data structures used:

| | |
|---|---|
| Entity *adapterQueryEntity* | An entity, which holds all the attributes corresponding to the client's query request with access modifiers for accessing the attributes. |
| List *adapterResults* | List of adapter components matching the query requirements. |

Table 4.4. Data Structure for AM functions

### 4.5.1.1 Algorithm for AM Initialization

The AM on initialization creates the AM_REPOSITORY where the adapter component service information is stored. It activates the client request handler, which services client's query requests for adapter components and the adapter registration handler process, which receives calls from remote services to register them with the AM.

```
AM_INITIALIZATION
        CREATE AM_REPOSITORY
        ACTIVATE AM_CLIENT_REQUEST_HANDLER
        ACTIVATE AM_SERVICE_REGISTRATION_HANDLER
END_AM_INITIALIZATION
```

### 4.5.1.2 Algorithm for Handling Client Requests for Adapters

The AM receives natural language-like query requests for adapter components from clients. The process of parsing the query and generating a structured query language statement is similar to that used in the QM (Algorithm *QM_PROCESS_XML_QUERY* and *QE_GENERATE_SQL_QUERY*). The AM executes the resultant SQL query against the AM Repository and obtains the list of adapter components matching the search criteria.

```
AM_CLIENT_REQUEST_HANDLER
        IN: naturalLanguageAdapterQuery
        OUT: adapterResults

        WHILE TRUE
                IF contacted by client with REQUEST for adapter components
                        // Refer algorithms QM_PROCESS_XML_QUERY and
```

```
                    // QE_GENERATE_SQL_QUERY
                    sqlQuery = PROCESS naturalLanguageAdapterQuery
                    adapterResults = EXECUTE QUERY sqlQuery on
                            AM_REPOSITORY
                    return adapterResults to Client
                ENDIF
        ENDWHILE
END_AM_CLIENT_REQUEST_HANDLER
```

## 4.5.1.3 Algorithm for Registering Adapter Components

The AM receives the XML based specification of the adapter components and parses these specifications to extract the data and stores this information in the AM_REPOSITORY.

```
AM_ADAPTER_REGISTRATION_HANDLER
        IN: adapterComponentSpecification
         WHILE TRUE
                IF contacted by adapter component with REQUEST for registration
                    STORE adapterComponentSpecification in AM_REPOSITORY
                ENDIF
        ENDWHILE
END_AM_ADAPTER_REGISTRATION_HANDLER
```

## 4.6 Headhunters

Another critical component of URDS is a Headhunter. The Headhunters perform the following tasks: a) Service Discovery: detect the presence of service providers (Exporters), b) register the functionality of these service providers, and c) return a list of service providers to the ICB that matches the requirements of the consumer (Importers) requests forwarded by the QM.

The service discovery process performs the search based on multicasting. Once deployed in the UniFrame environment, the Headhunters periodically ($TP_{mcast}$ ms - time period between successive multicast cycles) multicast their presence (location address of the Headhunter) to a multicast group. The multicast group address is obtained from the DSM. The active registries (extended native registries), which also obtain a multicast group address from the DSM, listen for multicast messages from Headhunters on these multicast groups. When Active Registries receive a multicast message from a Headhunter with its location they respond to the message by unicasting their location information to the Headhunter. The Headhunters maintain a cache of the pairs *<registry address, $T_r$ (time-stamp of receipt)>*. The Headhunter uses the registry location information received to query the Registries for the component information of service providers they contain in order to register the service information details with itself. During the registration, the Headhunter stores into the meta-repository all the details of the service providers, including the UniFrame specifications. It uses this information during a matching-making process where it tries to find services that satisfy the computational, cooperational, auxillary attributes and QoS metrics specified in the search query. A component may be registered with multiple Headhunters. The functionality of Headhunters makes it necessary for them to communicate with Active Registries belonging to any model, implying that the cooperative aspect of Headhunters be universal.

The two main issues that need to be handled by the Headhunter apart from the stated functions are:

- *Failure Detection:* Failure detection (FD) involves keeping track of service exporter components which may no longer be active in the system for various reasons including voluntary withdrawal from their respective registries, network failure, node failure, etc. The failure detection in the Headhunter is done at the level of detecting failure of the active registries, which hold the service exporter components. The Headhunter keeps track of the time at which it obtains registry location information from various active registries. At regular time intervals ($TP_{purge}$ ms – time period between successive purge cycles) the Headhunter notes the 'freshness' of the information it holds and purges the registry information, which it deems to be 'stale'. 'Fresh' or 'Stale' are determined based on the time elapsed between the receipt of the registry address through unicast communication ($T_r$) and the current time ($T_c$). If the elapsed time is greater than 2 purge cycles (i.e. $(T_c - T_r) > 2TP_{purge}$) then it signifies that the registry is not responding and may be dead or 'stale', else the registry information is deemed to be 'fresh'. This is essentially based on the principle that if a registry is still active in the system it will respond to the Headhunter with its location information and thus have a later timestamp. A registry which for whatever reason is unable to contact the Headhunter with its information will hold a 'stale' timestamp and it will be assumed that all service exporter components held by this registry are no longer available for rendering service.
- *Multicast Security:* This involves securing the multicast data transmission mechanism from security threats such as eavesdropping, and masquerading. The Headhunter uses Secret-Key Encryption to ensure security of transmitted data. The secret-key used is a symmetric key wherein the sender and receiver use the same key for purposes of encryption and decryption.

### 4.6.1 Algorithms for HH Functions

Data structures used:

| | |
|---|---|
| Hash table *registryTable* | Mapping between Active Registry location addresses and the timestamp of when this location address information was received by the Headhunter. |
| Hash table *componentTable* | Mapping between Component IDs and components holding the detailed UniFrame Specification. |

Table 4.5. Data Structure for HH functions.

#### 4.6.1.1 Algorithm for HH Initialization

The Headhunter is configured at startup by the deployer with the DSM location information, domain name, username, and password. The Headhunter upon initialization contacts the DSM with its authentication credentials in order to obtain a multicast address and secret-key for its domain. The Headhunter then creates the meta-repository, joins the multicast group and starts the processes for sending multicast communication and failure detection, which execute periodically. It also activates the process to receive unicast communication, which receives calls from remote registries.

```
HH_INITIALIZATION
    IN: DSMLocation
    // Obtain handle to DSM which runs at well-known location
    dsm = LOOKUP DSMLocation
    /* Obtain an authenticated packet by contacting the DSM. Pass the authentication
    information (userType, userName, password, contactLocation, domain) to the
    DSM and wait for response */
```

```
        authenticatedPacket =
        CALL DSM_AUTHENTICATION_SERVICE on dsm with
        userType, userName, password, contactLocation, domain
        //Extract multicast address and secret-key from authenticatedPacket.
        multicastAddress = GET multicastAddress from authenticatedPacket
        secretKey = GET secretKey from authenticatedPacket
        CREATE META_REPOSITORY
        // Once multicast address is obtained join the multicast group.
        JOIN multicast group with multicastAddress
        START HH_SEND_MULTICAST_COMMUNICATION
        START HH_FAILURE_DETECTION
        ACTIVATE HH_RECEIVE_UNICAST_COMMUNICATION
END_HH_INITIALIZATION
```

### 4.6.1.2 Algorithm for Headhunter Multicast Announcements

This algorithm outlines the process for periodic encrypted multicast announcements by the Headhunter.

```
HH_SEND_MULTICAST_COMMUNICATION
        WHILE TRUE
                /* Keep multicasting the encrypted contact information of this Headhunter
                at regular intervals to the multicast group address, which has
                subscriptions from listeners, which are the active registries. */
                encryptedHeadhunterLocation = ENCRYPT_secretKey{HeadhunterLocation}
                MULTICAST encryptedHeadhunterLocation
                // Put the multicasting thread to sleep for TP_mcast millisecs before the next
                // multicast.
                SLEEP TP_mcast
        ENDWHILE
END_HH_SEND_MULTICAST_COMMUNICATION
```

<u>4.6.1.3 Algorithm for Receiving Unicast Communication from Active Registries</u>

In this algorithm, the Headhunter receives the registry location from active registries. It then computes the timestamp of receipt and stores the registry location and timestamp in a hash table. It then uses the registry location information it has received to contact the registry and retrieve component information available with the registry to store in the meta-repository.

```
HH_RECEIVE_UNICAST_COMMUNICATION
        IN: registryLocation
        WHILE TRUE
                IF contacted by registry with registryLocation
                        /* Compute the timestamp when information was obtained
                        The timestamp is computed as the number of millisecs elapsed
                        since January 1st, 1970 GMT. */
                        $T_r$ = COMPUTE CURRENT_TIMESTAMP
                        /* Add the registry location information and timestamp to the
                        registry table.*/
                        PUT <registryLocation, $T_r$>in registryTable
                        CALL HH_POPULATE_META_REPOSITORY with
                        registryLocation
                ENDIF
        ENDWHILE
END_HH_RECEIVE_UNICAST_COMMUNICATION
```

#### 4.6.1.4 Algorithm for Populating Meta Repository

This algorithm demonstrates how the Headhunter gets a handle to the active registries using their registry location and contacts the registries to retrieve the component information, which it stores onto the meta-repository.

```
HH_POPULATE_META_REPOSITORY
    IN: registryLocation
    // Get handle to active registry using its registryLocation
    activeRegistry = LOOKUP registryLocation
    // Obtain the component data stored in this registry.
    componentTable = CALL AR_GET_COMPONENT_DATA on activeRegistry
    // Store the component information from the componentTable into the Meta-
    // Repository
    WHILE componentTable HAS MORE ELEMENTS.
        componentInfo = GET NEXT ELEMENT from componentTable
        STORE componentInfo to META_REPOSITORY
    ENDWHILE
END_HH_POPULATE_META_REPOSITORY
```

#### 4.6.1.5 Algorithm to Retrieve Search Results from the Meta-Repository

This algorithm outlines the process in which the Headhunter generates the SQL query from the *queryEntity* and executes this query against the meta-repository to retrieve the list of components matching the search criteria.

```
HH_EXECUTE_QUERY
    IN: queryEntity
    OUT: resultTable
    sqlQuery = CALL QE_GENERATE_SQL_QUERY on queryEntity
    resultTable = EXECUTE QUERY sqlQuery on META_REPOSITORY
    RETURN resultTable
END_HH_EXECUTE_QUERY
```

### 4.6.1.6 Algorithm to Detect Failure of Active Registries

This failure detection algorithm of the Headhunter involves keeping track of active registries which may no longer be alive. The Headhunter records the timestamp of responses received from these registries and periodically checks the freshness of the registry timestamps purging a reference to all the 'stale' registries and also deleting all the component information obtained from them.

```
HH_FAILURE_DETECTION
    WHILE TRUE
        /* The Headhunter maintains time stamped entries of  responses
         received from the active registries. Periodically ($TP_{purge}$ millisecs) a
        check is performed to verify the 'currentness' of members */
        SLEEP $TP_{purge}$
        $T_c$ = COMPUTE CURRENT_TIMESTAMP
        registryList =GET enumeration of locations from registryTable
        WHILE registryList HAS MORE ELEMENTS.
            registryLocation = GET NEXT ELEMENT from registryList
            $T_r$ = GET timestamp in registryTable for registryLocation
            IF $(T_c - T_r) > 2TP_{purge}$
```

```
                DELETE components in repository obtained from this
                registryLocation
                REMOVE <registryLocation, $T_r$> from registryTable
            ENDIF
        ENDWHILE
    ENDWHILE
END_HH_FAILURE_DETECTION
```

<u>4.6.1.7 Algorithm for Headhunter Shutdown</u>

The Headhunter before shutdown withdraws from the DSM, leaves the multicast group and terminates all the active processes.

```
HH_SHUTDOWN
    // Withdraw Headhunter registration from DSM
    CALL DSM_WITHDRAW on dsm with hhLocation
    // Leave multicast group.
    LEAVE multicast group at multicastAddress
    // Stop all active threads of control..
    STOP HH_SEND_MULTICAST_COMMUNICATION
    STOP HH_FAILURE_DETECTION
HH_SHUTDOWN
```

## <u>4.7 Meta-Repositry (MR)</u>

The Meta-Repository is a data store that holds service information of components adhering to different models. The Meta-repository stores the *Meta* level service information comprising of: a) Service type name, b) Details of its informal specification, and c) Zero or more QoS values for that service for each of the components. The implementation of a Meta-Repository is database-oriented and all the component information is stored onto a database. Using a database implementation provides an opportunity to perform searches for components matching the search criteria by using the inbuilt searching techniques. The Meta-Repository is a *passive component,* i.e., a Headhunter brings information and stores it in the meta-repository.

## 4.8 Active Registry (AR)

The native registries (e.g., RMI Registry or CORBA registry) are extended to have the following features:

- *Activeness:* The registries are modified to be able to listen to multicast messages from the Headhunter and respond with their registry location information.
- *Introspection Capabilities:* The registries are extended to not only keep a list of component URLs of those components registered with them but also their detailed UniFrame specifications. This is achieved by querying the components (using principles of introspection) to obtain the URL of their XML based specifications. The registries parse the specification and maintain the details in a memory resident table, which is returned to the Headhunter upon request.

### 4.8.1 Algorithms for AR Functions

Data structures used:

| | |
|---|---|
| Hash table *componentTable* | Mapping between Component ID and component specification details. |
| Entity *componentEntity* | An entity, which can be persisted to a repository and holds all the attributes corresponding to the UniFrame specification with access modifiers for accessing the attributes. |

Table 4.6. Data Structure for AR functions.

#### 4.8.1.1 Algorithm for AR Initialization

The algorithm for AR initialization involves contacting the DSM for an authenticated packet of the multicast address and the secret-key. Using the multicast address the AR joins the multicast group and starts listening for multicast communication from the Headhunters on this multicast group.

```
AR_INITIALIZATION
    IN: DSMLocation
    // Obtain handle to DSM which runs at well-known location
    dsm = LOOKUP DSMLocation
    /* Obtain an authenticated packet by contacting the DSM. Pass the authentication
    information (userType, userName, password, contactLocation , domain) to the
    DSM and wait for response */
    authenticatedPacket =
    CALL DSM_AUTHENTICATION_SERVICE of dsm with
     userType, userName, password, contactLocation, domain
```

```
        //Extract multicast address and secret-key from authenticatedPacket.
        multicastAddress = GET multicastAddress from authenticatedPacket
        secretKey = GET secretKey from authenticatedPacket
        // Once multicast address is obtained join the multicast group.
        JOIN multicast group with multicastAddress
        START AR_RECEIVE_MULTICAST_COMMUNICATION
        ACTIVATE AR_GET_COMPONENT_DATA
END_AR_INITIALIZATION
```

4.8.1.2 Algorithm for AR Receiving Multicast Communication

This algorithm outlines the process in which an AR receives multicast communication from the Headhunters. On receiving the Headhunter multicast messages with its location information the AR decrypts the message using the secret-key and unicasts it's encrypted contact location to the Headhunter.

```
AR_RECEIVE_MULTICAST_COMMUNICATION
        IN: encryptedHeadhunterLocation
        WHILE TRUE
                IF contacted by Headhunter with encryptedHeadhunterLocation
                        /* Receive encrypted Headhunter location information multicast to
                        registry  by Headhunters. Decrypt the location information using
                        secret-key. */
                        HeadhunterLocation =
                         $DECRYPT_{secretKey}$ {encryptedHeadhunterLocation}
                        //Get a handle to the headunter running at the location
                        Headhunter = LOOKUP HeadhunterLocation
                        CALL HH_RECEIVE_UNICAST_ COMMUNICATION on
                        Headhunter with registryLocation
                ENDIF
        ENDWHILE
END_ AR_RECEIVE_MULTICAST_COMMUNICATION
```

### 4.8.1.3 Algorithm for Obtaining UniFrame Specifications of Registered Components

This algorithm outlines the process for obtaining the UniFrame specifications of the components registered with an AR. The AR gets a URL list of all the components registered with it. It then steps through this list and gets a handle to each of these components. Using the component reference, the AR examines the component's properties to check for a property returning the URL of its UniFrame specification. The AR then reads the XML-based UniFrame specification from the URL and parses this specification to obtain all the component details, which it stores in an entity object *componentEntity*. The AR builds a hash table of such entities corresponding to each of the components registered with it and returns this hash table to the Headhunter.

```
AR_GET_COMPONENT_DATA
        OUT: componentTable
        WHILE TRUE
                IF contacted by HH to retrieve component data
                        CREATE a new componentTable
                        //Obtain a list of object URL's of all objects registered with this
                        // registry.
                        registeredServicesURLList = GET LIST of service components
                                                        registered with this Active Registry
                        // For each object in this URL list
                        FOR i=0 to LENGTH of registeredServicesURLList
                                registeredServiceURL = registeredServicesURLList[i]
                                // Lookup and obtain the reference to the services from the
```

```
                    // registry using the registered service URL.
                    serviceObject = LOOKUP registeredServiceURL
                    // Obtain the location (URL) of the UniFrame Specification
                    // for this service by introspecting its property name called
                    // "uniFrameSpecification".
                    uniFrameSpecURL =
                    CALL AR_INTROSPECT_PROPERTY with
                     serviceObject, "uniFrameSpecification"
                    // Parse the UniFrame Specification and construct a
                    // componentEntity which can be persisted.
                    CREATE a componentEntity
                    document = PARSE uri and load XML document
                    CALL AR_PARSE_UNIFRAME_SPEC with document
                    //Add the component to the componentTable
                    PUT < registeredServiceURL, componentEntity>
                     in componentTable
                ENDFOR
                RETURN componentTable
            ENDIF
        ENDWHILE
END_AR_GET_COMPONENT_DATA
```

## 4.8.1.4 Algorithm for Parsing the UniFrame Specification

This algorithm uses recursion to parse through the nodes of the XML tree, extract the node values and store these values in the *componentEntity*. The algorithm starts parsing at the root node element. It extracts the node name and checks if the node name matches any attribute in *componentEntity* and populates it with the value in this node. It then finds all the children of that node and repeats the process through recursion.

```
AR_PARSE_UNIFRAME_SPEC
    IN: nodeElement
    nodeName = GET NODE NAME from nodeElement
    FOR each attribute in componentEntity
        IF nodeName EQUALS attribute
        nodeValue = GET NODE VALUE from nodeElement
            SET attribute value in componentEntity to nodeValue
        ENDIF
    ENDFOR
    // Get the list of children for this Node.
    NODELIST childrenList = GET CHILDNODES for nodeElement
    IF childrenList NOT NULL
        // For every child node in the list
        FOR i = 0 to LENGTH of childrenList
            childNode = childrenList[i]
            // Recurse through the function READ_NODE passing it the
            // childNode as reference.
            CALL AR_PARSE_UNIFRAME_SPEC with childNode
        ENDFOR
    ENDIF
END_AR_PARSE_UNIFRAME_SPEC
```

### 4.8.1.5 Algorithm for Introspection of the Registered Components

This algorithm outlines the process for examining a service object to find a specific property and retrieve its value. The algorithm gets a list of all the properties from the service object and tries to find a match for the specific property of interest. Once the property is found, a handle to the Read accessor method (getter) of this property is obtained and invoked.

```
AR_INTROSPECT_PROPERTY
    IN: serviceObject, propertyName
    OUT: property
    //Introspect and retrieve all information pertaining to this service object.
    serviceObjectInfo = INTROSPECT serviceObject to retrieve object information
    //Get a description list of all properties of this object.
    propertyDescriptorList =
    GET PROPERTY DESCRIPTORS from serviceObjectInfo
    //Iterate through the description list to find the desired property.
    FOR i=0 to LENGTH of propertyDescriptorList
        propertyDescriptor = propertyDescriptorList[i]
        propertyDescriptorName = GET NAME of propertyDescriptor
        //If the property descriptor name matches the desired property name
        IF propertyDescriptorName EQUALS propertyName
            //Get the accessor method that returns the property value.
            method = GET READ METHOD of propertyDescriptor
            //Invoke the method to retrieve the value.
            property = INVOKE method of serviceObject
            //Return this property to the requester.
            RETURN property
        ENDIF
    ENDFOR
END_AR_INTROSPECT_PROPERTY
```

## 4.9 Services

Service Exporter Components are implemented in different models, e.g., Java RMI, CORBA, EJB, etc. The components are identified by their *Service Offers* comprising of service type name, b) informal UniFrame specification, and c) zero or more QoS values for that service. A component registers its interfaces with an Active Registry. The component interface contains a method, which returns the URL of its informal specification. The informal specification is stored as an XML file adhering to certain syntactic contracts to facilitate parsing. These service exporter components will be tailored for specific domains such as Financial Services, Health Care Services, Manufacturing Services, etc., and will adhere to the relevant standards, business architectures, and research and technologies for these industry specific markets.

This chapter presented the high-level design for the components in the URDS architecture. The next chapter presents a prototype implementation for the URDS architecture.

# 5. IMPLEMENTATION OF THE URDS ARCHITECTURE

Chapter 4 provided a conceptual perspective of the URDS architecture. The architecture presented did not adhere to any specific implementation methodology. The URDS architecture can be realized using several different software/hardware technologies. This chapter describes a prototype implementation for the URDS architecture using Java and Java based technologies.

The chapter is organized as follows: Section 5.1 presents technological artifacts and architectural models, which were used in the prototype implementation. Section 5.2 presents the prototype implementation followed by experimental results from the prototype in Section 5.3. Section 5.4 outlines implementation strategies that can be used to enhance the prototype implementation.

## 5.1 Technology

This section describes various architectural artifacts and technologies that have been leveraged in the implementation the URDS architecture.

### 5.1.1 Web Servers and Application Servers

A *Web Server* consists of computer hardware and software programs that serve up static content like HTML pages, images and documents to remote browsers accessing the web server. Web Servers are assigned IP addresses using which remote applications can access it using the HTTP protocol. Web Servers are also called HTTP servers. Examples for Web Servers include the IBM Http Server [IBMa], iPlanet Web Server Enterprise Edition [IPL], and Apache Http Server [APA].

*Application Servers* are software components that collaborate with Web Servers to return customized results (dynamic content) to a client's request. Examples of Application servers include IBM WebSphere Application Server [IBMb], WebLogic [BEA], and IIS [MIC]. Figure 5.1 illustrates the relationship between Browsers/Clients, Web Servers and Application Server. The browser clients communicate with the web servers using HTTP protocol. The web servers in turn communicate with the application servers to retrieve dynamic content, which is returned to the clients. The application servers communicate with back-end database systems/directory servers using standard communication protocols.

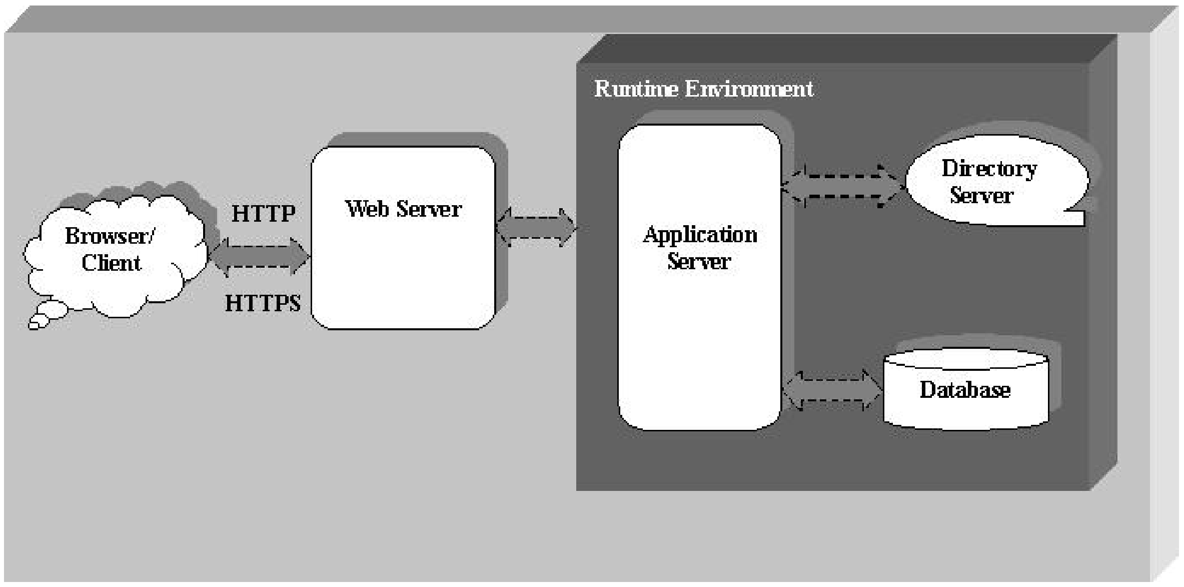


Figure 5.1. Interaction between Clients, Web Servers and Application Servers.

### 5.1.2 Java$^{TM}$ 2 Platform Enterprise Edition (J2EE$^{TM}$)

The prototype implementation is based on the architectural model laid out by [SUN01b] in J2EE$^{TM}$. J2EE$^{TM}$ defines a standard that applies to all aspects of architecting, developing, and deploying multi-tier, server-based applications. Figure 5.2 [SUN01b] shows the components of the J2EE Model.

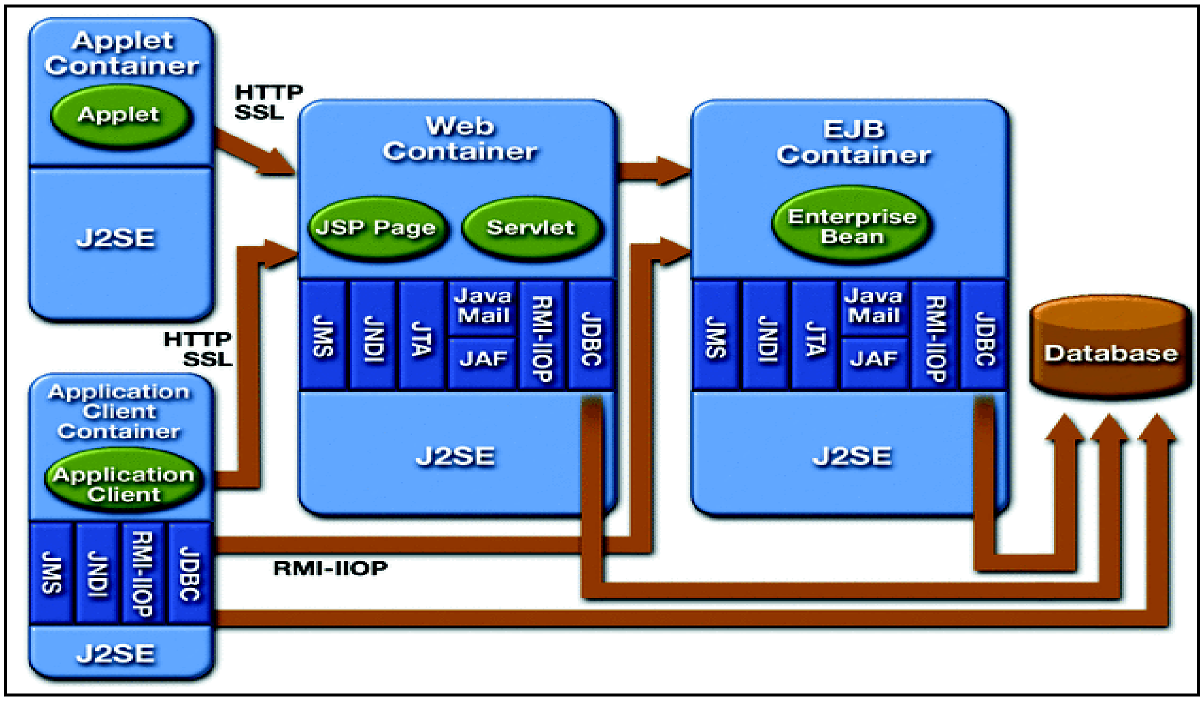


Figure 5.2. Components and Containers of J2EE Model.
(Figure 5.2 is courtesy SUN Microsystems, [SUNa])

The J2EE$^{TM}$ platform specifies technologies to support multi-tier enterprise applications. These technologies fall into three categories [SUN01b]: Component, Service, and Communication. The following sub-sections outline the technologies (see Figure 5.2) in each of these categories that have been used in the implementation. (The Java based technologies described in these sections are proprietary technologies of SUN Microsystems [SUN01b]).

### 5.1.2.1 Component Technologies

Components are application-level software units. All J2EE components depend on the runtime support of a system-level entity called a "*Container*". Containers provide components with services such as life cycle management, security, deployment, and threading.

The J2EE Component technologies have been used in the prototype to create the front-end client components and back-end service components. The prototype supports two types of clients: *Application Clients* which are structured as *Application Client Components* (described in section 5.1.2.1.1) and *Web Clients*, which interact with the *Web Components* (described in section 5.1.2.1.2). The prototype also uses the *JavaBean Components* (described in section 5.1.2.1.3) in the client and server tiers of the implementation.

The different types of J2EE components used in the prototype are described below:

#### 5.1.2.1.1 Application Client Components

*Application clients* are client components that execute in their own Java virtual machine. *Application clients* are hosted in a *Application Client Container*.

#### 5.1.2.1.2 Web Components

A *Web Component* is a software entity that provides a response to a request. The J2EE platform specifies two types of Web components: Servlets and JavaServer Pages$^{TM}$ (JSP) pages. A *Web Container* hosts web components.

- *Java Servlet Technology 2.3:* Servlets extend the capabilities of web servers that host applications accessed by way of a request-response programming model. Java Servlet technology has provisions for defining HTTP-specific servlet classes. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers.
- *JavaServer Pages Technology 1.2:* The JavaServer Pages (JSP) technology provides an extensible way to generate dynamic content for a Web client. A JSP

page is a text-based document that describes how to process a request to create a response. It contains two types of text: static template data, which can be expressed in any text-based format such as HTML, WML, and XML, and JSP elements (snippets of servlet code), which determine how the page constructs dynamic content.

#### 5.1.2.1.3 JavaBean and Enterprise JavaBean Components

A *JavaBean* Component is a body of code with fields and methods to implement modules of business logic. The server and client tiers can include components based on the JavaBeans component architecture. *Enterprise JavaBeans* are server-side components that encapsulate the business logic of an application and support enterprise level processing by making the JavaBeans scalable, transactional, and multi-user secure. An EJB container hosts Enterprise beans.

### 5.1.2.2 Service Technologies

The J2EE platform service technologies allow applications to access a variety of services. The prominent service technologies supported are *JDBC™ API 2.0* which provides access to databases, *Java Transaction API (JTA) 1.0* for transaction processing, *Java Naming and Directory Interface (JNDI) 1.2* which provides access to naming and directory services, and *J2EE Connector Architecture 1.0* which supports access to enterprise information systems.
The service technologies used in the prototype are described below:

- *JDBC™ API 2.0:* The JDBC™ API provides methods to invoke SQL commands from Java programming language methods. The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the J2EE platform.

- *Java API for XML Processing 1.1:* XML is a language for representing text-based data so the data can be read and handled by any program or tool. Programs and tools can generate XML documents that other programs and tools can read and handle. Java API for XML Processing (JAXP) supports processing of XML documents using DOM, SAX, and XSLT parsers. JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

#### 5.1.2.3 Communication Technologies

Communication technologies provide mechanisms for communication between clients and servers and between collaborating objects hosted by different servers. Some of the communications technologies supported by the J2EE Platform include - Transport Control Protocol over Internet Protocol (TCP/IP), Hypertext Transfer Protocol HTTP 1.0, Secure Socket Layer SSL 3.0, Java Remote Method Protocol (JRMP), Java IDL, Remote Method Invocation over Internet Inter ORB Protocol (RMI-IIOP), Java Message Service 1.0 (JMS), JavaMail and Java Activation Framework.

The prototype uses the *HTTP 1.0 Protocol* for communication between the browser-based clients and server side components. The inter-component communication on the server side is achieved through Java Remote Method Invocation.

The protocols used in the prototype are described below.

- *HTTP 1.0 Protocol:* The Hypertext Transfer Protocol (HTTP) is an application-level, generic stateless protocol for distributed, collaborative, hypermedia information systems.
- *Java Remote Method Protocol (JRMP):* Remote Method Invocation (RMI) is a set of APIs in the Java programming language that enables developers to build distributed applications. RMI uses Java language interfaces to define remote objects and a

combination of Java serialization technology and the Java Remote Method Protocol (JRMP) for performing remote method invocations.

## 5.2 Prototype Implementation

This section describes an implementation of a prototype for the URDS architecture described in Chapter 4. Figure 5.3 illustrates this implementation. The architecture is implemented as a multi-tier, distributed application model, which means

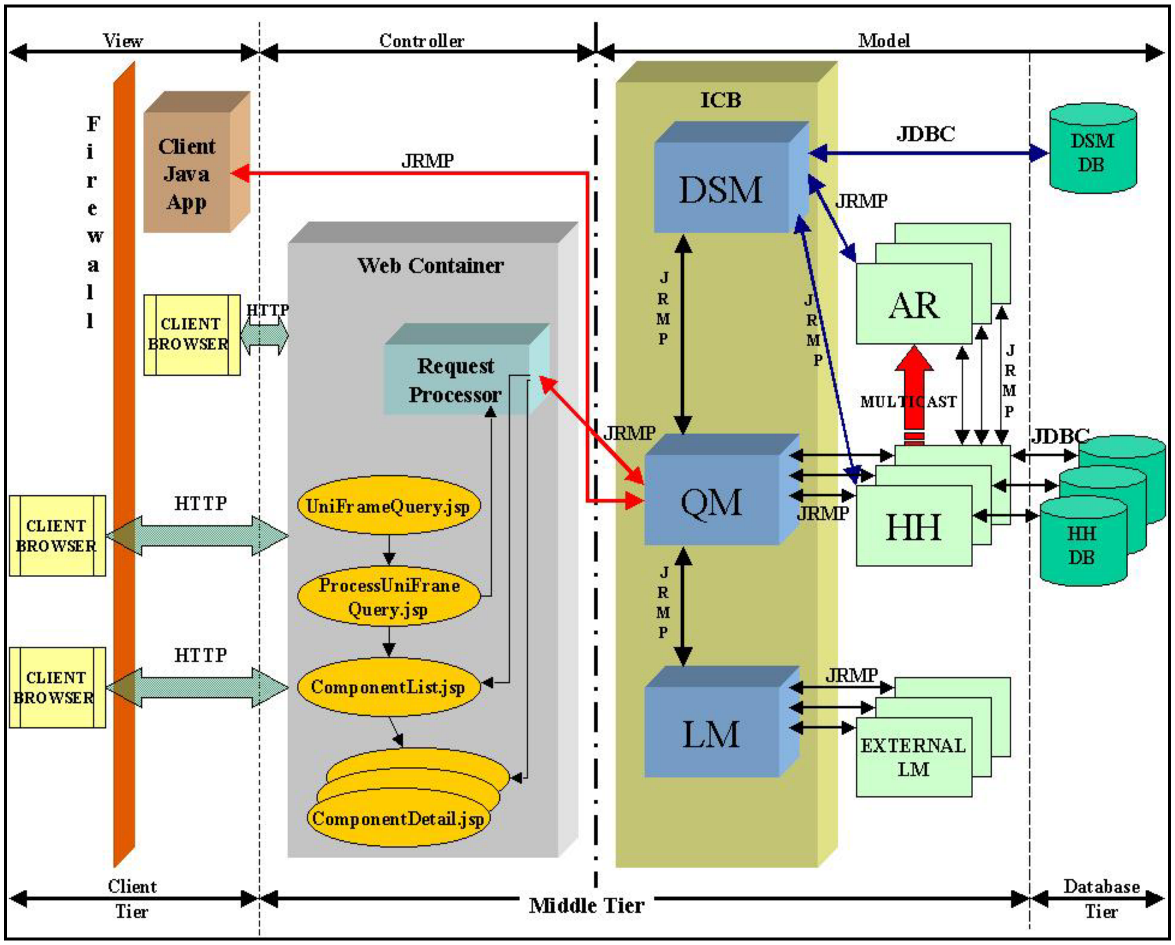


Figure 5.3. URDS Implementation

that the various parts of the prototype implementation can run on different machines. This is in conformance with the J2EE architecture, which defines a *client tier*, a *middle tier* (consisting of one or more sub tiers), and a backend tier providing services of enterprise information systems. In the prototype, the client tier supports a variety of client types, both application clients as well as web-based clients. The middle tier supports client services through Web containers in the *Web tier*. The middle tier also supports the component services (DSM, QM, LM, HH, AR which form the core of the URDS architecture) as Java-RMI based services. The *Database tier* supports access to the repositories by means of standard APIs.

### 5.2.1 Platform and Environment

In the prototype, the algorithms outlined for the various components are implemented using the Java$^{TM}$ 2 Platform, Standard Edition (J2SE) [SUNb] version 1.4 software environment. The core architectural components (DSM, QM, LM, HH, and AR) are implemented as Java-RMI based services. The repositories (DSM_Repository and Meta_Repositories) are database-oriented implementations based on Oracle v 8.0. The web-based components (JSPs), which service client interactions, are housed in the Tomcat 4.0 Servlet/JSP Container.

### 5.2.2 Communication Infrastructure

The unicast communication between the core architectural components is based on JRMP. The multicast communication between HHs and ARs is achieved through Multicast Sockets based on UDP/IP. The connections to the databases are established using JDBC APIs. Interactions between the clients (users) and the web components are based on HTTP protocol.

## 5.2.3 Security Infrastructure

The security infrastructure in the URDS utilizes the security and cryptography APIs that form a part of the Java$^{TM}$ Cryptography Architecture (JCA) [SUN02] and Java$^{TM}$ Cryptography Extension (JCE) frameworks [SUN00]. The specifications outlined in the JCA encompass the parts of the Java 2 SDK Security API related to cryptography. The JCE provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. The JCA and JCE frameworks are based on the concepts of "engine classes" and a "provider" architecture. An *engine class* defines a cryptographic service in an abstract fashion (without a concrete implementation). The actual implementations (from one or more providers) are those for specific algorithms. A "provider" refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key generation algorithms. The JCE Provider used in the prototype is the "SunJCE"[SUN00] which comes as a pre-installed and registered standard JCE Provider with the Java 2 SDK, v 1.4 release.

## 5.2.4 Programming Model

In the implementation of the prototype, the URDS functionality is partitioned into modules, and these modules are decomposed into specific objects to represent the behavior and data of the application. The prototype adapts the Model-View-Controller (MVC) architecture. The MVC architecture [GAM95, YOU95] can be described as: *"The **Model** represents the application data and the rules that govern access and modification of this data. The **View** renders the contents of a model. It accesses data from the model and defines how that data should be presented. The **Controller** defines application behavior; it translates user gestures into actions to be performed by the model".*

### 5.2.4.1 Implementation Details

In order to structure an application along the lines of the MVC architecture it is necessary to divide the application into objects and assign these objects to tiers. This process is referred to as object decomposition. Typically an application can be divided into three logical categories of objects: *"These are objects that deal with* presentation *aspects of the application, objects that deal with the application rules and data, and objects that accept and interpret user requests and control the application objects to fulfill these request"* [SUNa].

Following sections outline the object decomposition in the prototype along the lines of the MVC architecture and explain the functionality served by each of these objects in the prototype.

#### 5.2.4.1.1 The Model

This section explains the subdivision of components in the "Model" segment of the architecture. The components in this segment have been categorized as: *Entity Objects*, *Helper Objects*, *Persistent Data*, and the *Component Services* depending on their functionality.

##### 5.2.4.1.1.1 Entity Objects

Entity objects serve to represent the individual rows in a database as objects or to encapsulate an application specific concept in terms of an object. The entity objects in the prototype support accessor methods to set and retrieve the values of the attributes they hold. These objects can be passed by value as serializable Java objects. The prototype contains the following entity objects *ComponentBean*, *FunctionBean, QueryBean, AuthenticatedPacket* whose class diagrams are illustrated in Figure 5.4.

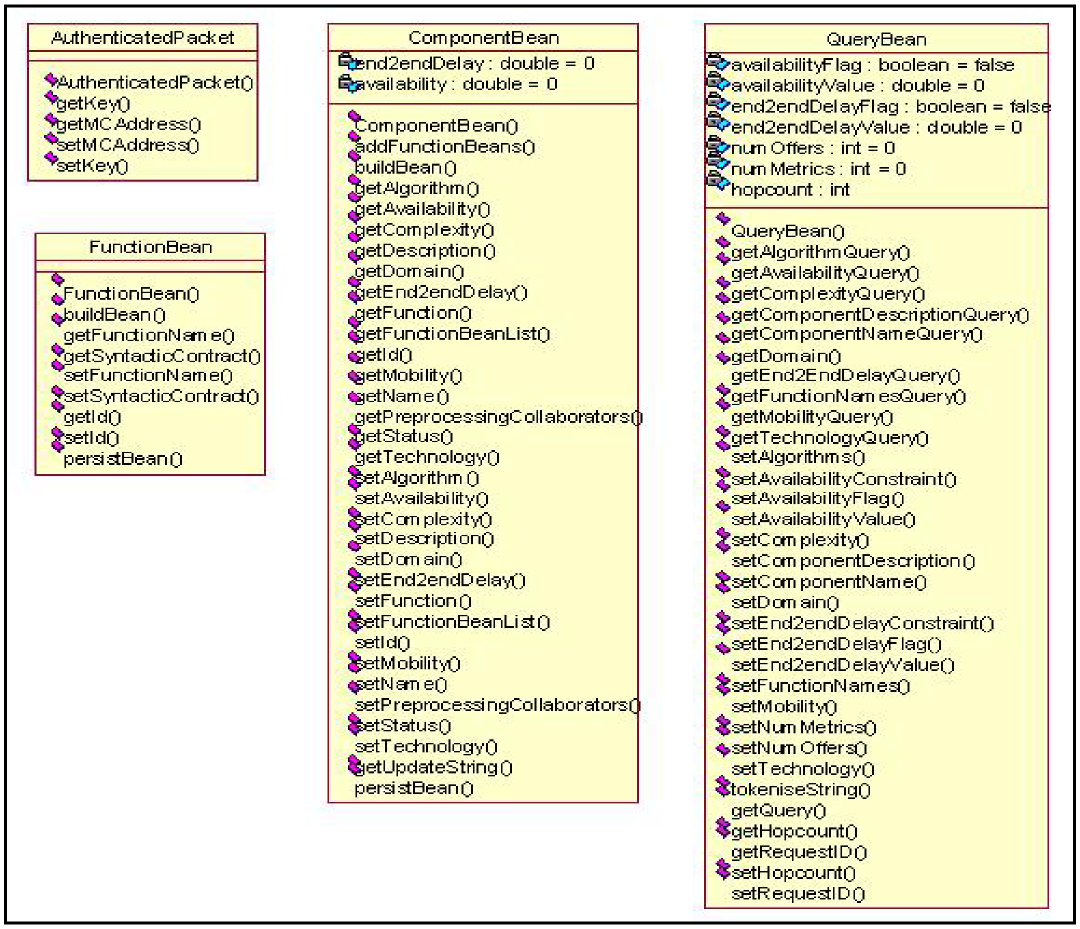


Figure 5.4. Class Diagrams for the URDS Entity Objects

Following is an explanation of these entity objects.

***ComponentBean:*** The attributes of the *ComponentBean* class mirror the fields of the table *Component* (see Section 5.2.4.1.1.3). The *ComponentBean* internally holds a list of *FunctionBean* objects. The *ComponentBean* has functionality built in to persist it to a database.

***FunctionBean:*** The attributes of the *FunctionBean* mirror the fields of the table *Function* (see Section 5.2.4.1.1.3). The *FunctionBean* has functionality built in to persist it to a database.

***QueryBean:*** The *QueryBean* encapsulates the attributes of a Query received from the client. The bean also has the logic associated with generating a SQL query based on the attributes it holds.

***AuthenticatedPacket*:** The *AuthenticatedPacket* encapsulates the multicast address and secret-key returned to the principals (Headhunters/ActiveRegistries) by the Domain Security Manager.

### 5.2.4.1.1.2 Helper Objects

The prototype uses Helper objects for purposes such as data access or for performing specific utility functions. The following classes serve as Helper classes and have been sub-classified under the categories of Data Access Objects and Dependent Objects.

#### 5.2.4.1.1.2.1 Data Access Objects

The data access objects used in the prototype encapsulate access to databases.

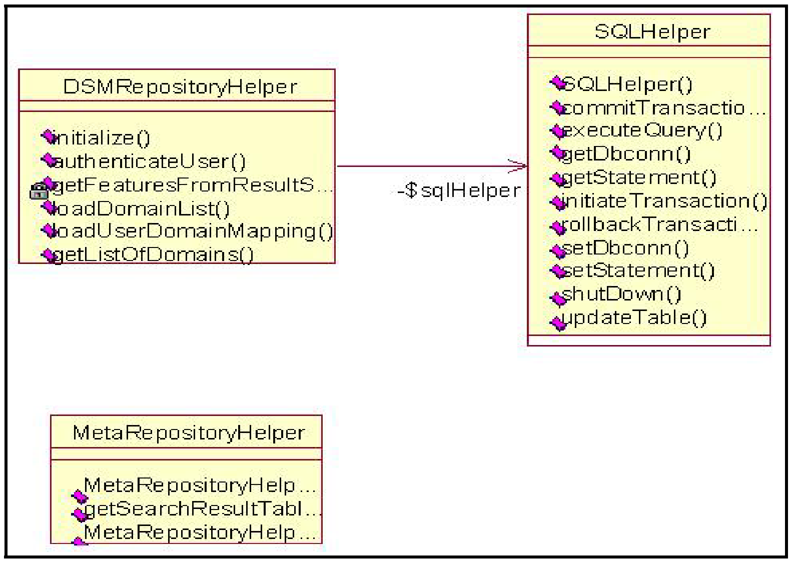


Figure 5.5. Class Diagrams for the Data Access Objects

Figure 5.5 illustrates the class diagrams of the data access objects. A description of the data access objects used in the prototype is provided here:

***DSMRepositoryHelper:*** This class performs functions associated with accessing the *DSM_Repository* to retrieve user-domain mappings and for user authentication.

***MetaRepositoryHelper:*** This class performs functions associated with accessing the *Meta_Repository* to retrieve search results.

***SQLEngine:*** This class acts as a wrapper, which encapsulates the essential, JDBC API methods and performs functions associated with establishing database connections and executing the queries. It is used by the *DSMRepositoryHelper* and *MetaRepositoryHelper* classes.

5.2.4.1.1.2.2 Dependent Objects

The prototype uses dependent objects for performing utility functions. These dependent objects are immutable and the objects that create and use them manage their life cycle.

Figure 5.6 illustrates the class diagrams of the dependent objects. A description of the dependent objects is provided below:

***CreateKey:*** This utility class is used to generate secret-keys. The keys are generated using a SUN provided engine class *javax.crypto.KeyGenerator*, which provides the functionality of a (symmetric) key generator. An instance of the javax.crypto.KeyGenerator class is obtained by specifying the Provider and the algorithm for key generation. The Provider used is the *"SunJCE"* an instance of which can be obtained by instantiating the *com.sun.crypto.provider.SunJCE* class. The algorithm specified is "DES" (Data Encryption Standard) [DES77].

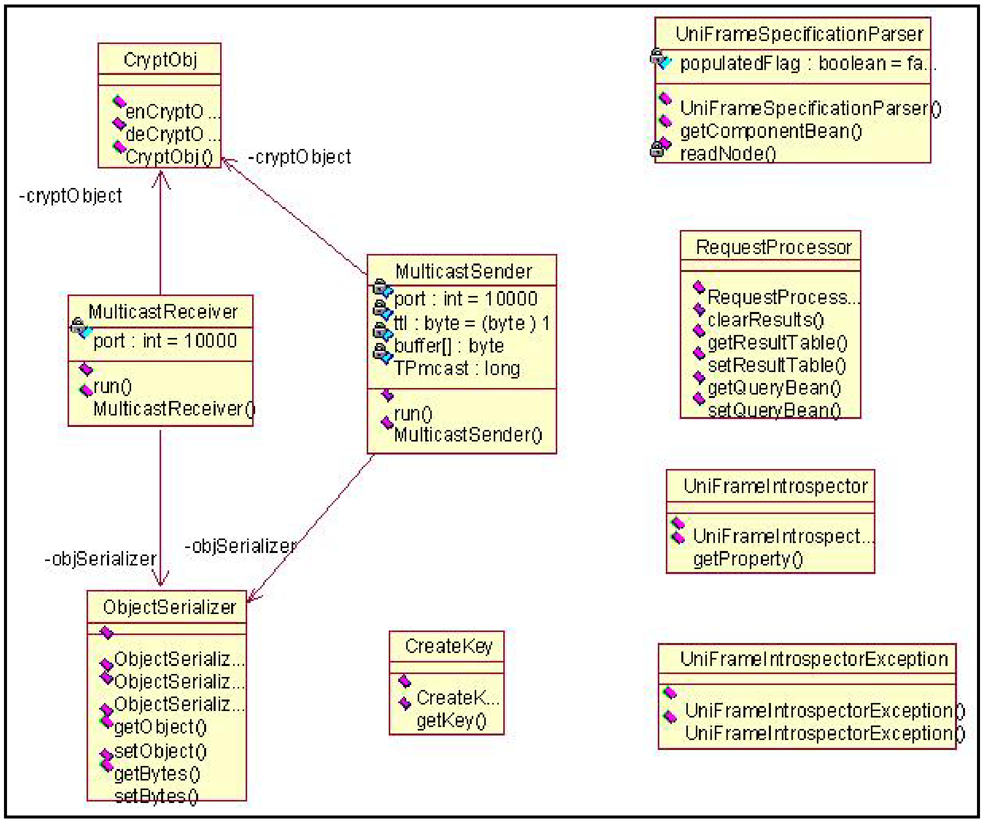


Figure 5.6. Class Diagrams for the Dependent Objects

***CryptObj:*** This class provides the utility functions for encrypting and decrypting data. The class utilizes a SUN provided engine class *javax.crypto.Cipher*, which is capable of carrying out encryption and decryption according to an encryption scheme (algorithm). An instance of the Cipher class can be obtained by specifying a *transformation* of the form "*algorithm/mode/padding*". The transformation used in the prototype is "*DES/ECB/PKCS5Padding*" where algorithm = *"DES"*, mode = *"ECB"* (Electronic Codebook Mode) [ECB80] and padding = "*PKCS5Padding"* (The Public Key Cryptography Standards #5)[RSA93]. Modes and padding schemes are present in the Cipher class because that class implements what is known as a block cipher; that is, it expects to operate on data one block (e.g., 8 bytes) at a time. Padding schemes are required in order to ensure that the length of the data is an integral number of blocks. Modes are provided to further alter the encrypted data in an attempt to make it harder to break the encryption. The *CryptObj* class encrypts an object with a cryptographic algorithm to create an instance of *javax.crypto.SealedObject*.

***ObjectSerializer:*** This utility class converts an object to byte array and reconstructs an object from byte array. To be used in secure multicast for serializing Sealed Objects.

***UniFrameIntrospector:*** This utility class uses reflection to analyze the properties of an object and retrieve the value corresponding to a specific attribute.

***UniFrameSpecificationParser:*** This class is used to parse a UniFrame XML specification file and construct an instance of the *ComponentBean*.

***MulticastSender:*** This class operates as a *Thread* which executes periodically. This *Thread* has a connection to a *MulticastSocket* to which it keeps multicasting *DatagramPackets* at regular intervals of time. This thread utilizes the *CryptObject* and *ObjectSerializer* Helper objects to multicast encrypted serialized messages.

***MulticastReceiver:*** This class operates on a Thread, which constantly listens for multicast messages on a *MulticastSocket* and receives the incoming *DatagramPackets*. This thread utilizes the *CryptObject* and *ObjectSerializer* Helper objects to decrypt and reconstruct the multicast messages received.

5.2.4.1.1.3 Persistent Data

The prototype maintains persistent data in database tables. These databases are associated with the service components (i.e. DSM and HH). The databases used in the prototype are the *DSM_Repository* and the *Meta_Repository*.

***DSM_Repository***

Figure 5.7 illustrates the schema for the *DSM_Repository*.

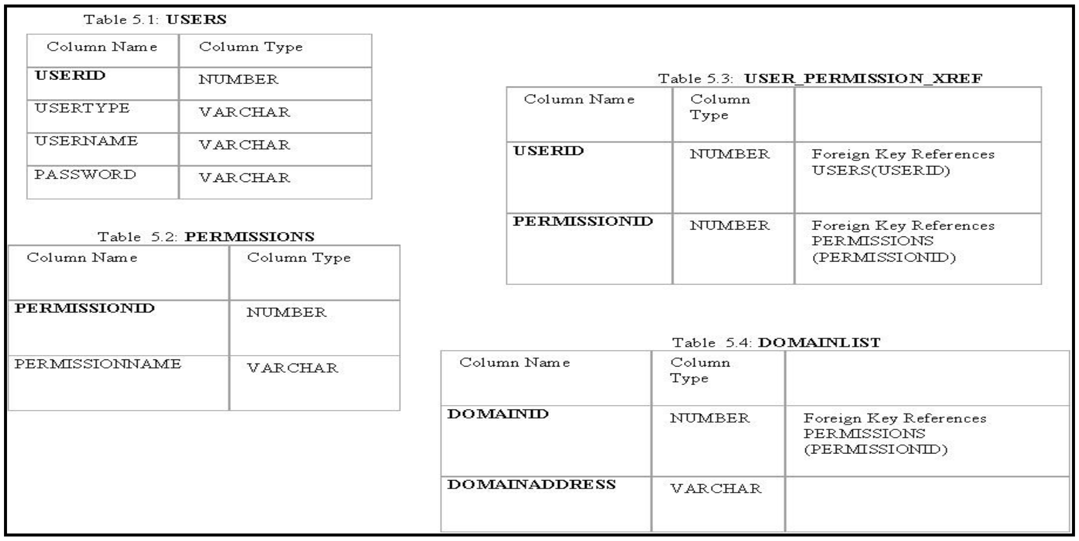


Table 5.1: **USERS**

| Column Name | Column Type |
|---|---|
| **USERID** | NUMBER |
| USERTYPE | VARCHAR |
| USERNAME | VARCHAR |
| PASSWORD | VARCHAR |

Table 5.2: **PERMISSIONS**

| Column Name | Column Type |
|---|---|
| **PERMISSIONID** | NUMBER |
| PERMISSIONNAME | VARCHAR |

Table 5.3: **USER_PERMISSION_XREF**

| Column Name | Column Type | |
|---|---|---|
| **USERID** | NUMBER | Foreign Key References USERS(USERID) |
| **PERMISSIONID** | NUMBER | Foreign Key References PERMISSIONS (PERMISSIONID) |

Table 5.4: **DOMAINLIST**

| Column Name | Column Type | |
|---|---|---|
| **DOMAINID** | NUMBER | Foreign Key References PERMISSIONS (PERMISSIONID) |
| **DOMAINADDRESS** | VARCHAR | |

Figure 5.7. DSM_Repository

The *DSM_Repository* comprises of four tables *Users*, *Permissions*, *User_Permission_Xref*, and *DomainList*.

The *Users* table serves to hold the Headhunter/ActiveRegistry information. The columns of this table are *userid* (numeric identifier), *usertype* (whether Headhunter or Registry), *username*, and *password*. The primary-key in this table is the *userid*. Example of a record of this table is as follows: <1,'Headhunter', 'Headhunter1', 'xxxx'>.

The *Permissions* table serves to hold a list of allowed permissions (or domains). The columns in this table are *permissionid* (numeric id for permission) and *permissionname* (domain name like Finance, Manufacturing, etc.,). The *permissionid* serves as the primary-key for this table. Example of a record of this table is as follows: <1, 'Manufacturing'>.

The *User_Permission_Xref* table serves to map the permission to be assigned to a user. The table contains two numeric columns *userid* which references the *userid* in *Users* and *permissionid* which references *permissionid* in *Permissions*. The combination of the <*userid,permissionid*> serve as the primary key. Example of a record of this table is as follows: <1, 2>.

The *DomainList* table serves to hold the list of multicast addresses and the domains that they map to. The table contains two columns *domainid,* which references *permissionid* of *Permissions* and *domainaddress,* which serves to hold the multicast address. The combination of the < *domainid, domainaddress* > serve as the primary key. Example of a record of this table is as follows: <1, '224.2.2.2'>.

***Meta_Repository***

Figure 5.8 illustrates the schema for the *Meta_Repository*.

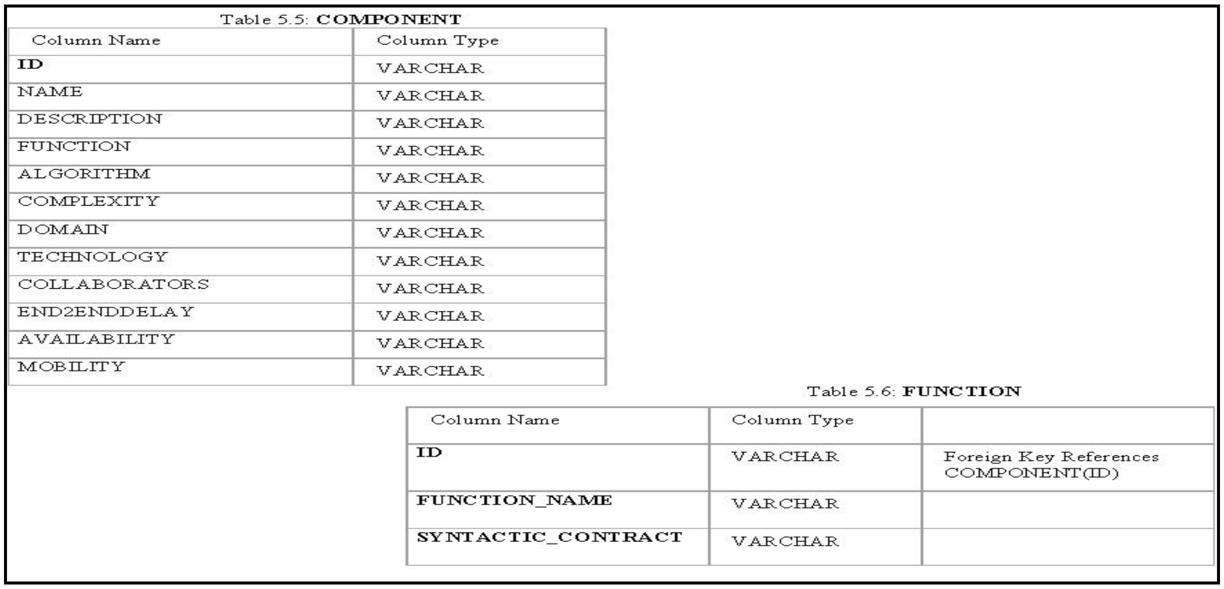


Table 5.5: **COMPONENT**

| Column Name | Column Type |
|---|---|
| **ID** | VARCHAR |
| NAME | VARCHAR |
| DESCRIPTION | VARCHAR |
| FUNCTION | VARCHAR |
| ALGORITHM | VARCHAR |
| COMPLEXITY | VARCHAR |
| DOMAIN | VARCHAR |
| TECHNOLOGY | VARCHAR |
| COLLABORATORS | VARCHAR |
| END2ENDDELAY | VARCHAR |
| AVAILABILITY | VARCHAR |
| MOBILITY | VARCHAR |

Table 5.6: **FUNCTION**

| Column Name | Column Type | |
|---|---|---|
| **ID** | VARCHAR | Foreign Key References COMPONENT(ID) |
| **FUNCTION_NAME** | VARCHAR | |
| **SYNTACTIC_CONTRACT** | VARCHAR | |

Figure 5.8. Meta_Repository

The *Meta_Repository* comprises of two tables *Component*, *Function*.

The *Component* table serves to hold the UniFrame Specification information of the components. The columns of this table are *id* (alpha numeric compinent id), and other variables which describe a component like *name*, *description, function, algorithm, complexity, domain, technology, collaborators,end2endDelay, availability, mobility*. The primary-key in this table is the *id*. Example of a record of this table is as follows: <'intrepid.cs.iupui.edu','AccountServer','Provides An Account Management System','Acts As An Account Server','Simple Addition And Subtraction','O(1)','Financial','Java-RMI','AccountClient',10,100,'No'>

The *Function* table serves to hold a list of all functions and the syntactic contracts . The columns in this table are *id* which references *id* of the *Component* table, *function_name* and *syntactic_contract*. The combination of <*id, function_name, syntactic_contract* > serves as the primary-key for this table. Example of a record of this table is as follows: <'intrepid.cs.iupui.edu','javaDeposit','void javaDeposit(float ip)'>

5.2.4.1.1.4 Service Components

A service component here refers to a software unit that provides a service. The service provided could be a computational effort or an access to underlying resources. A service component consists of one or more artifacts (software, hardware, libraries) that are integrated together to provide the service. Service Components are described by a remote interface and an implementation for this interface. Each Service Component is a stand-alone functional unit, which accepts inputs through its published interfaces, and returns results. Clients or other components can remotely access a service component using standard communication protocols. For a component to be remotely accessible, it should be deployed in a runtime environment (e.g. Application Server) with ports open to receive incoming requests and return results.

Figure 5.9 describes an example of a stand-alone remote accessible service component, which uses the HTTP protocol to accept and return results.

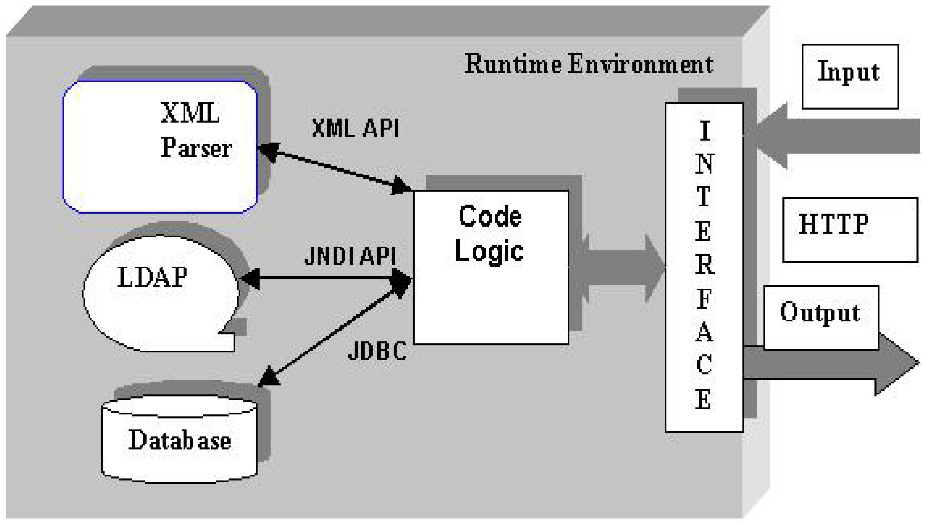


Figure 5.9. Example of a Stand-Alone Remote Accessible Service Component.

The service components implemented in the prototype are the DSM, QM, HH, and AR. These service components utilize the Entity Objects and Helper Objects to achieve their functionalities and store and retrieve information from their respective repositories.

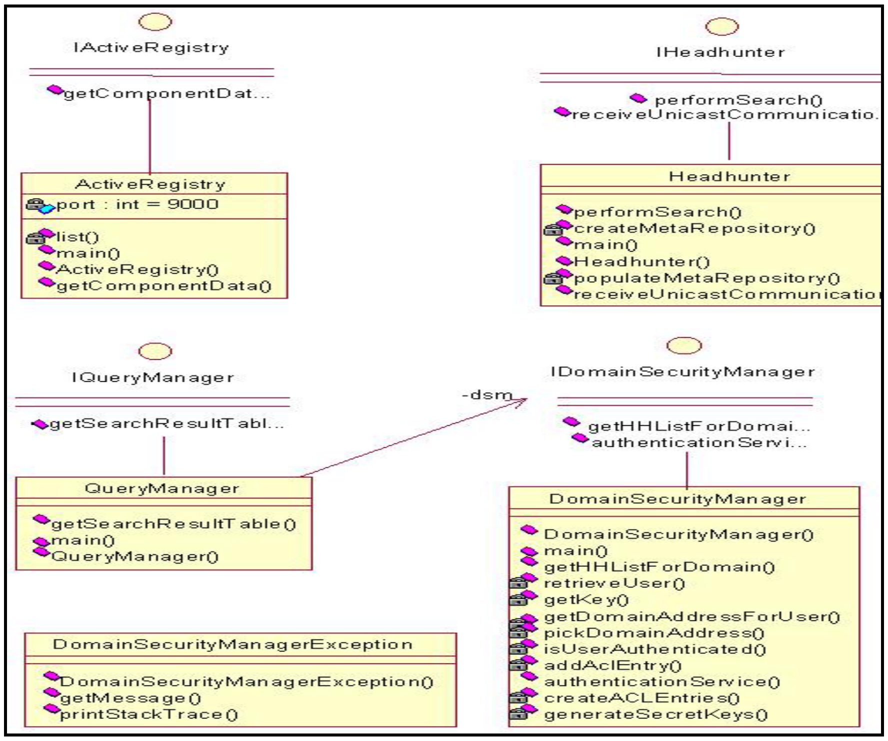


Figure 5.10. Class Diagrams for the Service Components

Figure 5.10 illustrates the class diagrams of the Service Components. The following sub-sections describe these service components.

#### 5.2.4.1.1.4.1 Domain Security Manager Service

*IDomainSecurityManager:* This is the interface for the DomainSecurityManager service. This interface publishes two methods:

- *getHHListForDomain*: This method is invoked by the QueryManager. The purpose of this method is to return a list of registered Headhunters for a particular domain to the QueryManager.
- *authenticationService*: This method is invoked by the Headhunter and ActiveRegistry components to authenticate themselves with the DSM.

*DomainSecurityManager:* This class implements *IDomainSecurityManager* interface. The implementation is based on the algorithms outlined for the DSM in Chapter 4. The

DSM performs the functions of generating secret-keys for the domains using the *CreateKey* helper class. It authenticates the principals against the *DSM_Repository* with the help of the *DSMRepositoryHelper* and distributes the secret-keys and multicast addresses to Headhunters and Active Registries. The communication between the parties is based on RMI-JRMP.

#### 5.2.4.1.1.4.2 Query Manager Service

*IQueryManager:* This is the interface for the *QueryManager* service. This interface publishes the following method:

- *getSearchResultTable*: This method is invoked by the *RequestProcessor* to obtain a list of services matching the search criteria specified by a component assembler.

*QueryManager:* This class implements the *IQueryManager* interface. The *QueryManager* class propagates the search query as a *QueryBean* object to the list of *Headhunter* components obtained from *DomainSecurityManager* and returns search results to the *RequestProcessor*. The interaction between these components is based on RMI-JRMP.

#### 5.2.4.1.1.4.3 Headhunter Service

*IHeadhunter:* This is the interface for the Headhunter service. This interface publishes the following methods:

- *performSearch*: This method is invoked by the *QueryManager* to propagate a search query.
- *receiveUnicastCommunication*: This method is invoked by the *ActiveRegistry* to inform the Headhunter of its location.

*Headhunter:* This class implements the *IHeadhunter* interface. The implementation is based on the algorithms outlined in the previous Chapter for the Headhunter functions. The Headhunter class interacts with the *QueryManager* and *DomainSecurityManager* using RMI-JRMP. The Headhunter uses the *MulticastSender* to multicast encrypted messages to *ActiveRegistry* services. The Headhunter also communicates with the ActiveRegistry services through unicast RMI-JRMP for the purpose of obtaining the component information as a *Hashtable* of *ComponentBean* objects. The HeadHunter saves this component information to the Meta_Repository and uses the *MetaRepositoryHelper* for the purpose of performing searches against the repository. The SQL query for the searches is obtained from the *QueryBean* which is propagated to the *Headhunter* by the *QueryManager*.

###### 5.2.4.1.1.4 Active Registry Service

*IActiveRegistry:* This is the interface for the ActiveRegistry service. This interface publishes the following method:

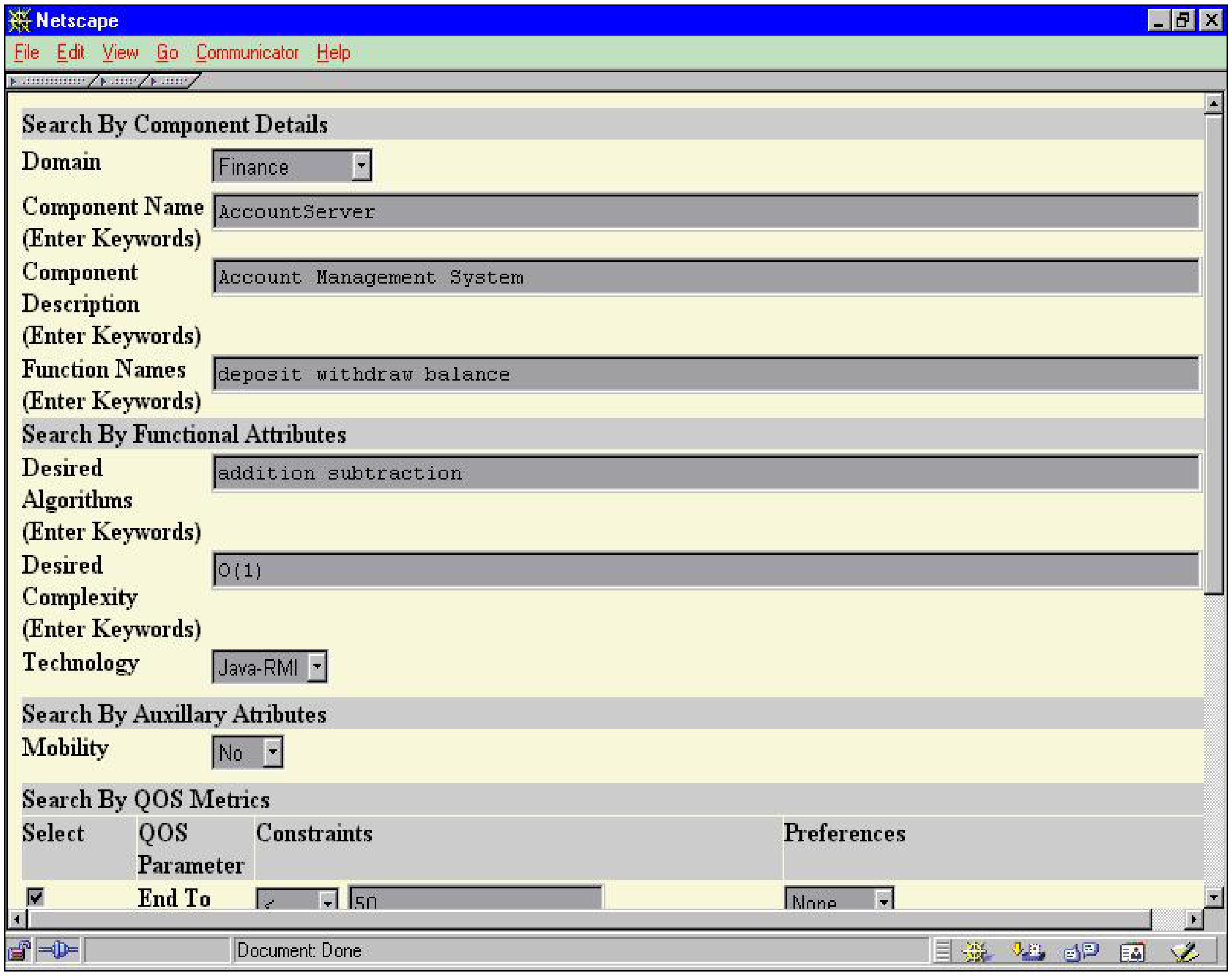
- *getComponentData:* This method is invoked by the *Headhunter* to retrieve component information from the *ActiveRegistry*.

*ActiveRegistry:* This class implements the *IActiveRegistry* interface. The implementation is based on the algorithms outlined in the previous Chapter for the *ActiveRegistry* functions. The *ActiveRegistry* class receives multicast messages from the *Headhunter* by using the *MulticastReceiver* class. Interaction with the *Headhunter* for passing its contact information and component information are done through RMI-JRMP. It also communicates with the *DomainSecurityManager* for authentication purposes using RMI-JRMP. The *ActiveRegistry* uses the *UniFrameIntrospector* for the purpose of obtaining the UniFrame specifications of the components registered with it. It also uses the *UniFrameSpecificationParser* for parsing the specifications and building instances of *ComponentBean* objects, which it returns to the *Headhunter*.

###### 5.2.4.1.2 The View

The view determines the presentation of the user interface of the prototype. The components that work together to implement the view are the JSP pages and the

JavaBeans components. The JSP pages are used for dynamic generation of HTML responses. JavaBeans components represent the contract between JSP pages and the model. JSP pages rely on these beans to read model data to be rendered to HTML. The following are the JSP pages, which present a View to the user: *UniFrameQuery.jsp* (see Figure 5.11 and Figure 5.12), *ComponentList.jsp* (see Figure 5.13), *ComponentDetail.jsp* (see Figure 5.14). The controller functionality associated with these views is explained in Section 5.2.4.1.3.



Search By Component Details

| | |
|---|---|
| Domain | Finance |
| Component Name (Enter Keywords) | AccountServer |
| Component Description (Enter Keywords) | Account Management System |
| Function Names (Enter Keywords) | deposit withdraw balance |

Search By Functional Attributes

| | |
|---|---|
| Desired Algorithms (Enter Keywords) | addition subtraction |
| Desired Complexity (Enter Keywords) | O(1) |
| Technology | Java-RMI |

Search By Auxillary Atributes

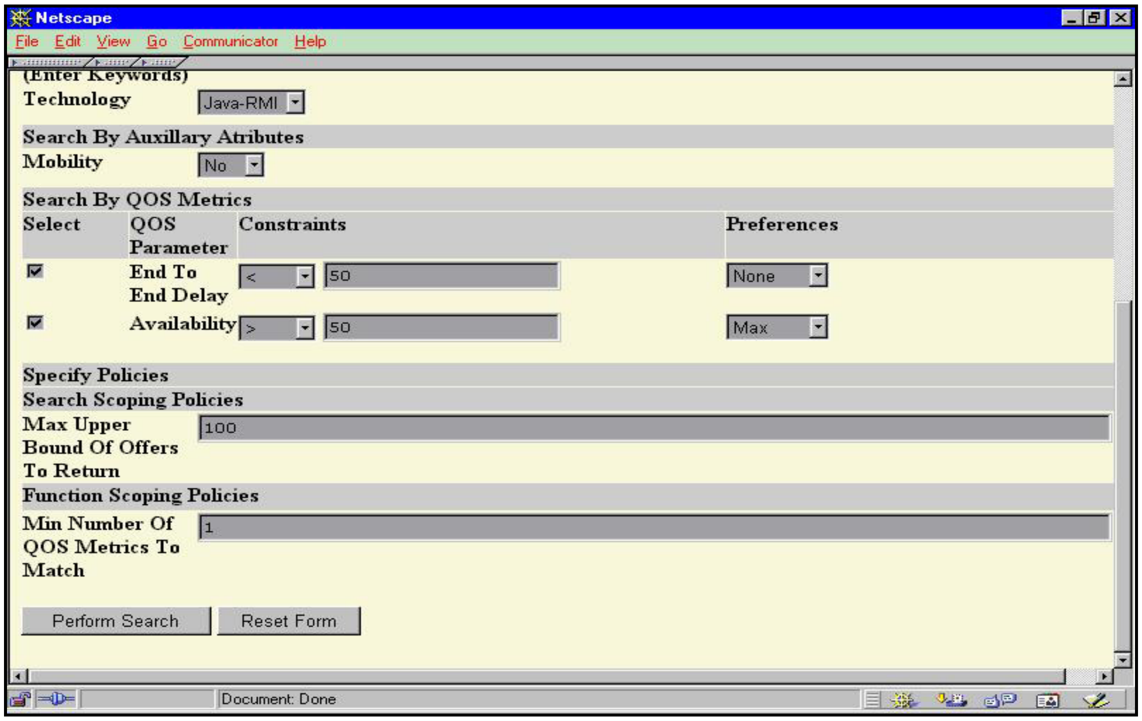
| | |
|---|---|
| Mobility | No |

Search By QOS Metrics

| Select | QOS Parameter | Constraints | | Preferences |
|---|---|---|---|---|
| ✓ | End To | < | 50 | None |

Figure 5.11 UniFrameQuery.jsp View

Figure 5.11 and 5.12 illustrate the UniFrameQuery.jsp view which allows the component assembler or system integrator to specify various search criteria.

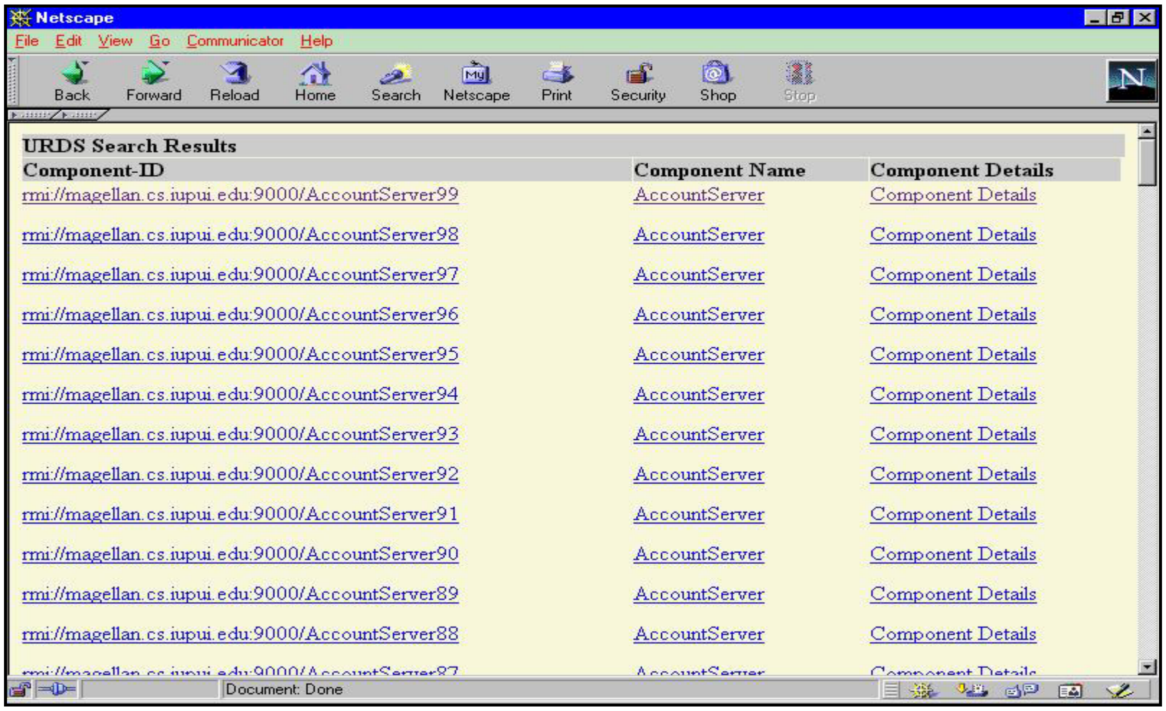


(Enter Keywords)

| | | | |
|---|---|---|---|
| Technology | Java-RMI | | |
| **Search By Auxillary Atributes** | | | |
| Mobility | No | | |
| **Search By QOS Metrics** | | | |
| Select | QOS Parameter | Constraints | Preferences |
| ✓ | End To End Delay | < 50 | None |
| ✓ | Availability | > 50 | Max |
| **Specify Policies** | | | |
| **Search Scoping Policies** | | | |
| Max Upper Bound Of Offers To Return | 100 | | |
| **Function Scoping Policies** | | | |
| Min Number Of QOS Metrics To Match | 1 | | |

Perform Search | Reset Form

Figure 5.12 UniFrameQuery.jsp View Continued.

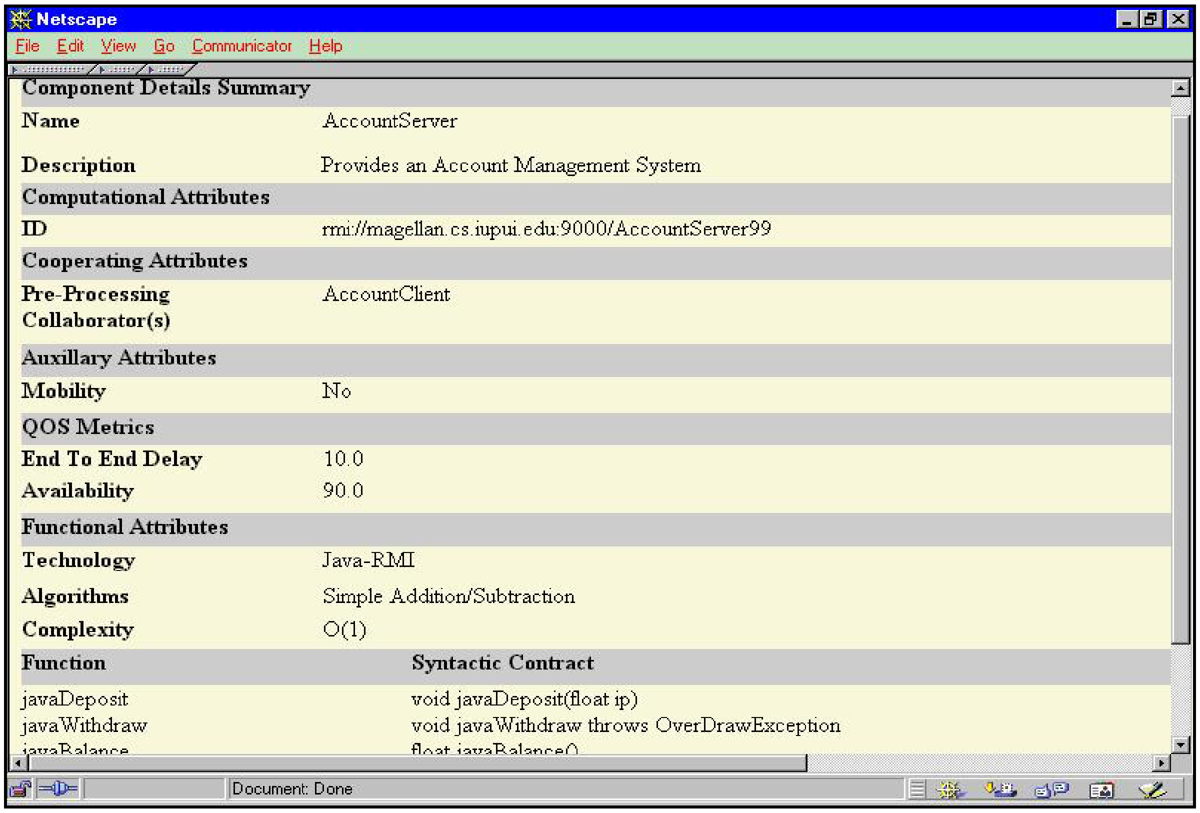


URDS Search Results

| Component-ID | Component Name | Component Details |
|---|---|---|
| rmi://magellan.cs.iupui.edu:9000/AccountServer99 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer98 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer97 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer96 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer95 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer94 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer93 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer92 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer91 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer90 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer89 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer88 | AccountServer | Component Details |
| rmi://magellan.cs.iupui.edu:9000/AccountServer87 | AccountServer | Component Details |

Figure 5.13 ComponentList.jsp View

Figure 5.13 above illustrates the view ComponentList.jsp. In this view the list of components matching the search criteria is displayed to the component assembler. The user can click on the Component Details hyperlink to view the details of a particular component which takes them to the next view ComponentDetails.jsp illustrated in Figure 5.14. The ComponentDetails.jsp view provides all the information specified by the component developer in the UniFrame specification of the component.



| Component Details Summary | |
|---|---|
| Name | AccountServer |
| Description | Provides an Account Management System |
| **Computational Attributes** | |
| ID | rmi://magellan.cs.iupui.edu:9000/AccountServer99 |
| **Cooperating Attributes** | |
| Pre-Processing Collaborator(s) | AccountClient |
| **Auxillary Attributes** | |
| Mobility | No |
| **QOS Metrics** | |
| End To End Delay | 10.0 |
| Availability | 90.0 |
| **Functional Attributes** | |
| Technology | Java-RMI |
| Algorithms | Simple Addition/Subtraction |
| Complexity | O(1) |
| **Function** | **Syntactic Contract** |
| javaDeposit | void javaDeposit(float ip) |
| javaWithdraw | void javaWithdraw throws OverDrawException |
| javaBalance | float javaBalance() |

Figure 5.14 ComponentDetail.jsp View

5.2.4.1.3 Controller

The prototype must reflect the state of a user's interaction with the discovery service and the current values of persistent data in the user interface. Following the MVC architecture, this functionality is implemented within the controller. The controller is responsible for coordinating the model and view. The following controller components provide this functionality in the prototype:

***UniFrameQuery.jsp:*** Receives and processes HTTP requests. The servlet generated from *UniFrameQuery.jsp* receives all HTTP requests.

***ProcessUniFrameQuery.jsp***: Processes the information from the HTTPRequest to create an instance of *QueryBean* and passes it to the RequestProcessor, which coordinates all handling of the request.

***RequestProcessor:*** *RequestProcessor* provides the glue in the Web tier for holding the application components together. It contains logic that needs to be executed for each request. *RequestProcessor* collaborates with the *QueryManager* and gets a *Hashtable* of search results from the *QueryManager*, which it forwards to the View *ComponentList.jsp.*

***ComponentList.jsp:*** Provides a *master view* of the lists of components matching the search criteria.

***ComponentDetail.jsp***: Provides a *detail view* that describes the details of a particular component. Users click on an item in the master view to zoom in on details, including a description, and other details of the UniFrame specifications.

#### 5.2.4.1.3.1 Managing the State of a Session

Every client session needs to track the information associated with the client requests (client query) and the associated responses (results matching the search criteria). In the JSPs an HTTP session object maintains the JavaBeans that are specific to a client. The following state information is maintained.

*Query Information:* The information associated with the client query is captured in the *QueryBean* and passed on to the *QueryManager* via the *RequestProcessor*.

*The Query Results:* The *RequestProcessor* class maintains a list of results matching the search criteria. The result list is stored as a *Hashtable* of *ComponentBean* objects within the *RequestProcessor* class. These results are displayed in the *ComponentList.jsp* view to the users.

## 5.3 Experimental Results

Some experimental tests were conducted to validate the performance of the prototype. The Service Components in the experimental setup comprised of one ICB (having a QueryManager and a DomainSecurityManager), one Headhunter and one ActiveRegistry (extended RMI registry). A single client (Application Client Component) was used in the experiment. The experimental tests were conducted with queries having different types of search constraints.

The measurements presented are averaged over 100 trials and were conducted on Sun Solaris machines running Sun Solaris Unix v5.8. Sun's JDK 1.4 was used to run the components of the URDS system.

The following performance metrics were gathered:

*Average Authentication Time ($T_{auth}$)*: This is computed as the time taken by the DomainSecurityManager to authenticate a principal. It is computed from the time a Headhunter/ActiveRegistry sends the DSM its authentication credentials to the time it receives a response from the DSM.

*Average Query Service Time ($T_{query}$):* This is the average time taken to service a query. It is computed from the time a query is presented by a client to the URDS system to the time a response is received.

*Average Registry Discovery Time($T_{discovery}$ ):* This is the average time taken by a Headhunter to discover an ActiveRegistry. It is computed from the time the Headhunter multicasts its location to the time it receives a response from the Registry. This time includes the time taken by the ActiveRegistry to decrypt the encrypted multicast communication received from the Headhunter.

*Average Component Information Retrieval Time ($T_{info\text{-}retrival}$):* This is the average time taken by the Headhunter to retrieve component information from an ActiveRegistry. It is computed from the time the Headhunter requests an ActiveRegistry for component information to the time it receives a response from the ActiveRegistry. This time includes the time taken by an ActiveRegistry to obtain the UniFrame Specification URLs from the list of components registered with it and to parse the XML specifications to extract the component information.

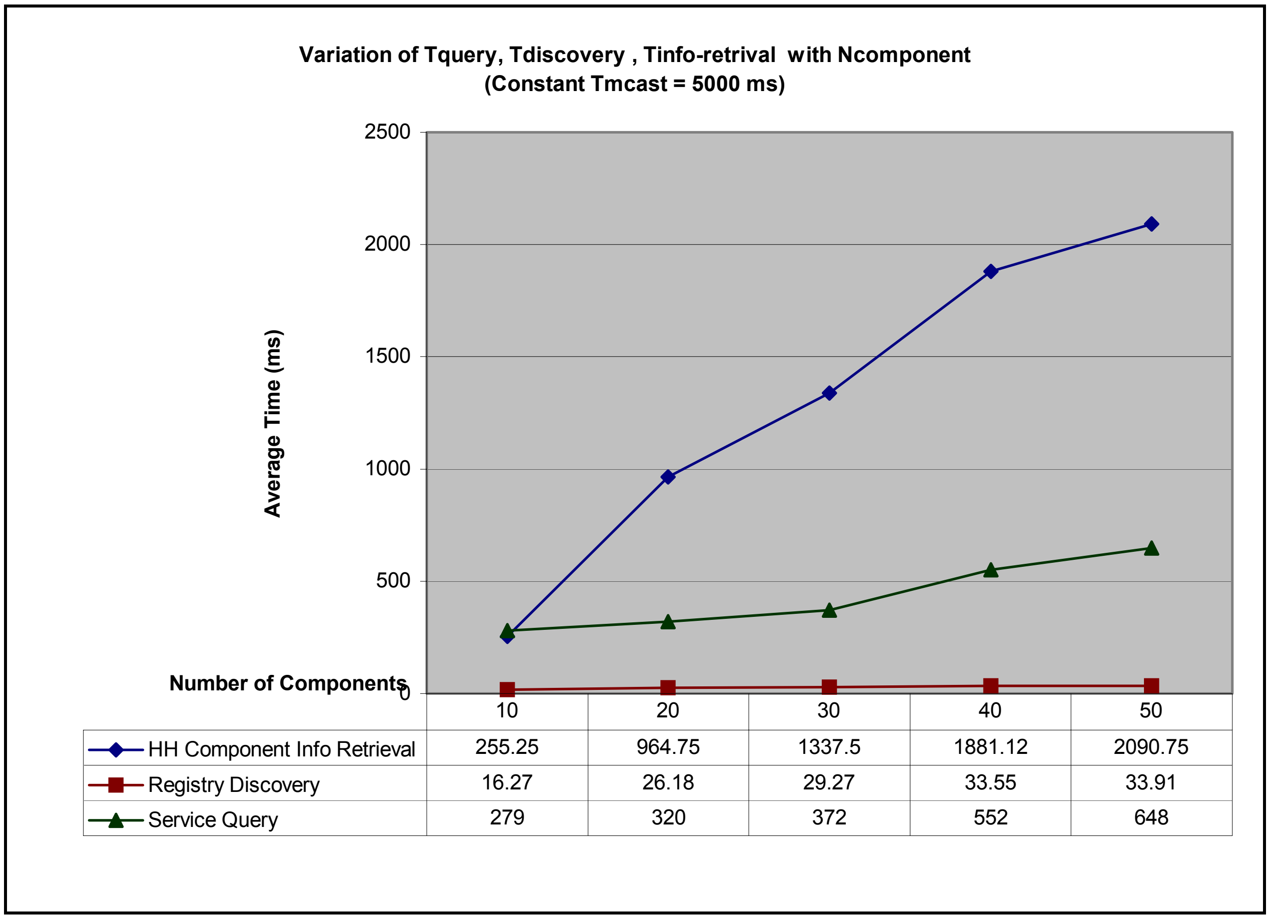
The Average Authentication Time was computed to be = 690.5 millisecs.

The Average Query Service Time, Average Registry Discovery Time, and Average Component Information Retrieval Time were calculated for the following two cases:

Varying the Time period between successive multicast cycles of the Headhunter ($T_{mcast}$) keeping the Number of Components ($N_{components}$) registered with the Active Registry constant.

Varying the Number of Components ($N_{components}$) registered with the Active Registry keeping the Time period between successive multicast cycles of the Headhunter ($T_{mcast}$) constant.

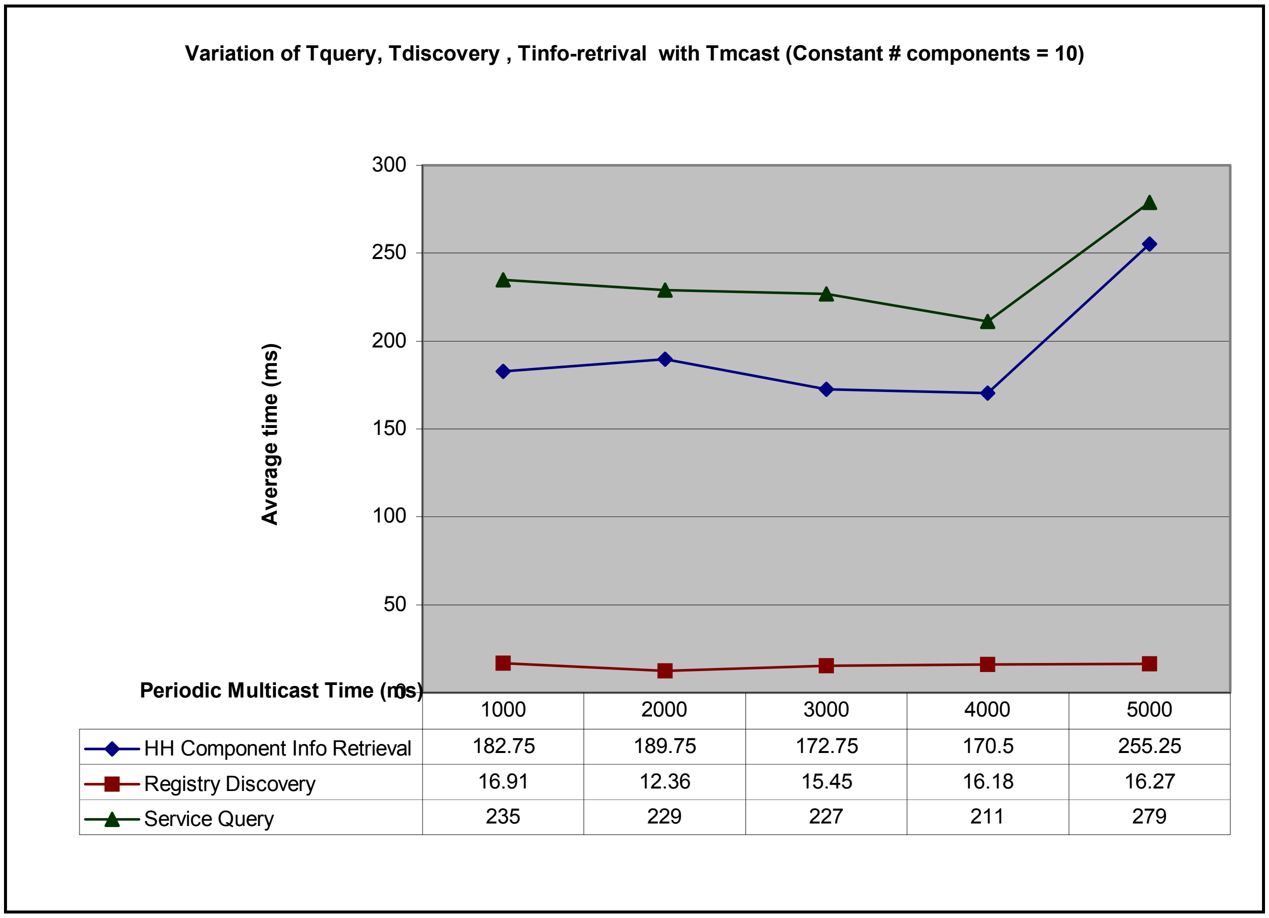
Figure 5.15 shows the variation of $T_{query}$, $T_{discovery}$ , $T_{info\text{-}retrival}$ with $N_{component}$ when $T_{mcast}$ was kept constant at 5000 ms. It can be observed from the graph that the average time taken the Headhunter to discover registries and the average time taken to service a query increase marginally with an increasing number of registered components. However, the time taken by the Headhunter to retrieve component information from the registries increases substantially with increasing number of registered components.



| Number of Components | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| ◆ HH Component Info Retrieval | 255.25 | 964.75 | 1337.5 | 1881.12 | 2090.75 |
| ■ Registry Discovery | 16.27 | 26.18 | 29.27 | 33.55 | 33.91 |
| ▲ Service Query | 279 | 320 | 372 | 552 | 648 |

Figure 5.15 Variation of $T_{query}$, $T_{discovery}$ , $T_{info\text{-}retrival}$ with $N_{component}$

Figure 5.16 shows the variation of $T_{query}$, $T_{discovery}$ , $T_{info\text{-}retrival}$ with $T_{mcast}$ when $N_{component}$ was kept constant at 10. It can be observed from the graph that the average time taken the Headhunter to discover registries remains almost constant with increase in time period between successive multicasts. The average time taken to service a query and the time taken by the Headhunter to retrieve component information from the registries also do not show much variation until the time period between multicast cycles is increased to 5000 ms, when they show an increase in the time taken.



| Periodic Multicast Time (ms) | 1000 | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|---|
| ◆ HH Component Info Retrieval | 182.75 | 189.75 | 172.75 | 170.5 | 255.25 |
| ■ Registry Discovery | 16.91 | 12.36 | 15.45 | 16.18 | 16.27 |
| ▲ Service Query | 235 | 229 | 227 | 211 | 279 |

Figure 5.16. Variation of $T_{query}$, $T_{discovery}$ , $T_{info\text{-}retrival}$ with $T_{mcast}$

## 5.4 Implementation Strategies for Enhancing the Prototype

This section discusses implementation strategies, to enhance the prototype at an application level and make the implementation more scalable, fault tolerant, maintainable, interoperable and secure. The strategies outline alternative deployment environments or communication protocols that can be used to implement the service components. The following are the performance areas in the implementation where these strategies are applicable:

*Improving Scalability and Fault Tolerance of the Implementation*

The prototype application supports various services like DSM, QM, HH, etc., It is desirable that these services be able to handle a large number of requests for services without their efficiency being adversely affected. Services that are scalable and easy to maintain usually have better QoS ratings. It would also be desired that these services be "available" for service most of the time and are not "unavailable" due to reasons such as hardware/software failure or "downtime" due to maintenance upgrades of the software/hardware units they are deployed in.

A suggested implementation for realizing these requirements is to deploy the Service Components in a runtime environment such as an Application Server, which supports the concept of Workload Management. An example of such an application server that implements Workload Management capabilities is IBM's WebSphere Application Server 4.0 [IBM02].

Workload Management (WLM) is the process of spreading multiple requests for a service over resources that can accomplish the task. WLM balances request processing by allowing incoming work requests to be distributed across application servers, containing identical versions of the service called 'Clones', according to a configured WLM selection policy. The software/hardware, which dispatches requests to the different servers in the pool, is called an "IPSprayer". Workload management is most effective when used in systems that contain servers on multiple machines. It can also be used in systems that contain multiple servers on a single, high-capacity machine. This enables the system to make the most effective use of the available computing resources.

If a server in the pool is down due to reasons such as hardware/software failure or maintenance upgrades then the requests for the services can be directed to another server in the pool containing the service clone. WLM improves the performance, scalability and reliability of an application by providing for load balancing and failover capabilities.

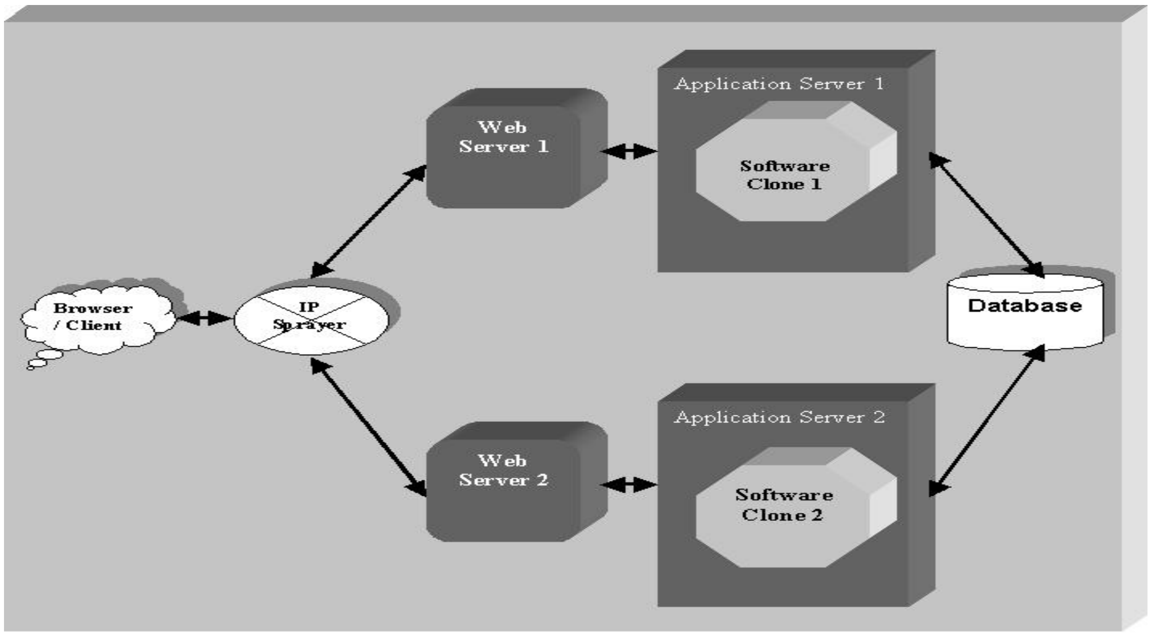


Figure 5.17. Improving Scalability with WLM

Figure 5.17 illustrates WLM across two servers.

*Interoperability and Asynchronous Communication*

In the prototype implementation, the service components such as DSM, QM, LM, HH, AR are implemented as Java-RMI based services which communicate with each other via JRMP. Alternative implementations could be to structure these services as Enterprise JavaBeans deployed in the EJB container communicating with each other via RMI-IIOP or as SOAP based services communicating with each other via SOAP. These protocols promote greater interoperability than JRMP. Using SOAP for inter-component communication removes the tight coupling that currently exists between the service components. Additionally, SOAP, being a firewall-friendly protocol, will remove the restrictions present in the current implementation of the services having to be located within the same subnet. This is especially a necessity in the case of the *LinkManager* as this will enable federation with external *LinkManager* components (*LinkManagers* in other ICBs) located in different networks.

All inter-component communication in the implementation is synchronous. The users are waiting on the QM to return results and the QM is waiting on the Headhunters to return results. Making these communications asynchronous and using remote event notification will allow a user greater flexibility wherein the user need not wait until the operation completes and results are returned. The use of asynchronous messaging allows the development of loosely-connected systems. These systems are typically more resilient in the event of failures, and more easily extensible as new applications are developed. Additionally, messaging provides an effective means of transmitting events between applications. Asynchronous messaging can be incorporated into the implementation using Java Message Service (JMS). The service components can be structured as EJB components, which integrate with JMS thus allowing the enterprise beans to participate fully in loosely connected systems. The EJB service components can then asynchronously notify other components of the occurrence of events. Remote event notification features will allow the Model to notify the Controller of changes in events in the model, which can be rendered as Views in the system.

### *Securing the Implementation*

In the current prototype implementation users of the URDS discovery services are not authenticated and there is no gradation on the levels of access that the users may possess. However, on a going forward basis, this would be a desired feature, as service providers may not wish to advertise their services to unauthorized users, or depending on the privileges the users possess, they may allow access to only a certain set of functionality as opposed to others. The authentication aspect for users can be handled through a form based userid/password scheme. For supporting users with different profiles, structuring the service components as EJB components allows the implementation to leverage the role based security services offered by the EJB architecture.

This chapter presented an implementation for the URDS architecture and the experimental results. The chapter also discussed enhancements that could be made to the prototype.

The next chapter concludes the thesis by presenting a comparison between the URDS architecture and other resource discovery protocols discussed in Chapter 2. The chapter also discusses possible future extensions to this work.

# 6. CONCLUSION

This thesis presented the architecture and an implementation for a resource discovery service, the “UniFrame Resource Discovery Service”. The resource discovery service itself forms a part of a framework, “UniFrame”, which aims at providing a platform for building DCS by integrating existing and emerging distributed component computing models under a common meta-model that enables discovery, interoperability, and collaboration of components via generative software techniques. Section 6.1 discusses the features of URDS and compares it to other resource discovery services. Section 6.2 discusses possible future enhancements to the URDS architecture.

## 6.1 Features of URDS

A discussion of the features of URDS and how these compare with other resource discovery protocols is presented here:

### 6.1.1 Interoperability

Various resource discovery protocols available today adhere to different “standards” and there is little interoperability between them. These protocols have been designed to provide services for very specific models. For example, the CORBA Trader services offer directory services only for CORBA objects, or the JINI discovery services spontaneously discover only other JINI enabled devices. For these protocols to be logically compatible, they need to be mapped by implementing equivalent protocols or bridges and proxies. This defies the underlying goal of “discovery” being universal. It is this interoperability gap that URDS is trying to bridge by providing discovery and directory services for components developed using diverse distributed computing models.

URDS tackles the issue of non-uniformity with the assistance of the ICB and Headhunter components (explained in Chapter 3 and Chapter 4). The Headhunters discover heterogeneous components. Since the components discovered by the Headhunters do not share common semantics for communication, the ICB provides the capability to generate the glue and wrappers for components implemented in diverse models to collaborate across the Internet. The ICB will eventually use standard component mappings and component model adapters to bridge the conceptual model disparity. Thus, URDS addresses the issue of bootstrapping between components implemented in diverse architectures, which is not found in other resource discovery protocols.

The Headhunter and ICB components together are responsible for allowing a seamless integration of different component models and sustaining (managed) cooperation among heterogeneous components.

### 6.1.2 Discovery Mechanism

It is essential that Discovery Services effectively manage the bandwidth usage. It is easy to saturate the network if the services implement protocols, which do not consider bandwidth as a critical issue (e.g., “if services and clients discover each other by multicast/broadcast once per second”). Protocols that are based on a purely advertisement or purely request strategy may use a great deal of bandwidth [GUT99]. If all services keep advertising their availability periodically and if clients keep sending service requests to all possible services, the network will saturate with advertisements and requests, and may also overload service providers. It is, therefore, imperative that multicast be used carefully and the number of messages exchanged be limited.

The fundamental difference between other Service Discovery Protocols and URDS is that, in the URDS architecture, the Clients and Services are not ‘active’ i.e., they do not directly participate in the multicast communications of the discovery process. It is the native registries that have a handle to the services that participate in the discovery process and not the services themselves. The advantage of this approach is two fold:

1. network usage is reduced as the Services and Clients are not participating in active discovery, which cuts down on the multicast messages periodically being broadcast/multicast by Services, Clients, and Lookup Services wanting to be discovered by each other.
2. service providers do not need to add additional functionality to their components to make them ‘active’ i.e., communication-aware. The components can be constructed in any desired component model and still be discovered in the URDS environment.

The URDS architecture of combining a Directory Service with a Discovery service, presents another unique advantage in reducing communication overhead. The ICB acts as the directory for brokering client requests. Clients contact the ICB with their service requests, and the ICB responds with a list of services matching the service requests. This avoids the situation of the client being swamped with responses from arbitrary services. URDS also supports Federation of the ICBs, which further helps to localize traffic and reduce the number of parties involved. The idea of Federation or Hierarchical organizations is common in Directory Services, and this can prove very efficient when combined with a Discovery service.

### 6.1.3 Service Description

Every resource discovery protocol specifies its own standards for describing services. The Object Management Group (OMG) specifies the standard Interface Definition Language (IDL) for defining service contracts. W3C has focused on standards for exchanging information using the Internet infrastructure (HTML, XML, HTTP, XSL). URDS aims to discover and achieve interoperability in an open, dynamic network environment where the communicating parties (clients and services) are implemented in diverse architectures each conforming to their own standards of service description. It was realized that it would not be possible to implement a meta-protocol of service types. The solution adopted in URDS is to allow the components to follow their own standards for service description but also provide a UniFrame specification in XML outlining the computational, functional, co-operational, auxiliary attributes and QoS metrics. Since, XML is designed to support an automatic translation and transformation between XML languages, it is well suited for providing interoperability between diverse services. The UniFrame Approach also provides capabilities for auto-generating the XML based UniFrame specifications from the natural language specification of the component.

### 6.1.4 Client Query and Matchmaking

In URDS, the users (clients) present a query to the system in natural language-like style. This query is passed through a natural language query processor, which extracts the key words, constraints, and preferences (see Chapter 4) from the query and constructs a structured query language statement. The matchmaking is achieved by comparing the structured query language statement against the repository of service information. This method of query and matchmaking allows a user a large degree of flexibility in formatting queries, which is not found in any of the other discovery or directory services. The JINI searching mechanism uses the Java serialized object-matching mechanism, which is based on exact matching of serialized objects, making it prone to false negatives due to class versioning problems [CZE99]. The SSDS searching mechanism is based on XML tag matching while SLP is based on matching of service types all of which require the client's query to adhere to a standard format in order to obtain a match.

### 6.1.5 Domain of Discovery

The discovery protocol of the URDS architecture is “administratively scoped”, i.e., it locates services within an administratively defined logical domain. *Domain* in the UniFrame Approach refers to industry specific markets such as Financial Services, Health Care Services, Manufacturing Services. In URDS, the Headhunters and Active-Registries are associated with specific domains and they register and detect services developed for the specific industrial sector with which they are associated. This kind of logical partitioning also allows for a contextualization of the search space. Other discovery protocols too have the notion of “administrative scope” but the difference in their case is that scope is associated with the topology of the network domain. The resources detected in these cases are those that are logically located ‘nearby’ within the network topology.

### 6.1.6 Security

Most Directory Services do not have a built-in security framework. Security is not as essential in these cases because of the centralized management schemes. Discovery protocols however need to address this issue because of their decentralized nature. Discovery protocols are faced with the challenge of providing a robust, yet lightweight, and automatic security protocol that does not consume network resources.

The UniFrame architecture addresses a majority of the security threats faced in the scenario of a discovery process. The security model provides for authentication of the principals involved (Headhunters and native registries) through a userid-password based authentication scheme. URDS uses access control lists to restrict access to multicast address resources. URDS also ensures the security of the multicast data transmitted through the use of a secret-key based data encryption scheme. The model does not at this stage provide for elaborate protocols for establishing keys and passwords. It uses a centralized controller (the Domain Security Manager), which is responsible for authenticating the principals and distributing the secret-keys.

The Ninja: SSDS, outlined in Chapter 2, is a good example as it uses several different security protocols and services. SSDS uses Certificates and a Certificate Authority structure to authenticate all principals. The authenticity of the discovery service is maintained through Public Key authenticated SDS server announcements. Privacy and authenticity of service descriptions are ensured through one-way encrypted service announcements (combined public and private key protocol). SSDS uses Secure RMI for a two-way authenticated and encrypted remote method invocation.

The other protocols like JINI and SLP have support for authentication, but no provisions for encryption.

### 6.1.7 Quality of Service

A unique feature of URDS is the incorporation of the notion of QoS as applied to software components. Although QoS parameters and associated metrics have been widely used in networking, there is no standard vocabulary for discussing QoS as it relates to distributed computing and component based solutions. The UniFrame research aims at outlining an approach to a QoS-based framework for creating distributed heterogeneous software components [RAJ01]. This work leverages work by Zinky, Bakken & Schantz [ZIN95] with the goal of providing a catalog of QoS parameters, indicating how parameters might be described, and providing a list and brief description of the QoS parameters being cataloged along with a detailed sample description. The discovery services of the UniFrame integrate this notion of QoS into the specification of services and the matchmaking process. The UniFrame specification of a component indicates the various QoS metrics supported by a component and the clients issue requests for components possessing desired levels of QoS. The matchmaking process then uses the

QoS metrics as an additional level of filtration to return the appropriate match to the client.

## 6.2 Future Work

This thesis proposed an architecture for discovering services developed in different component models. The proposed architecture was validated through a prototype. The prototype implements the following features of the URDS architecture: i) Dynamic Discovery of components, which are developed using the Java-RMI, distributed computing model, ii) Servicing Component Assembler's requests presented in natural language-like style, and iii) Security features outlined in the URDS architecture.

Future extensions to this research involve more comprehensive and complete implementation and a more rigorous validation of the features proposed in the URDS architecture, which are not currently implemented in the prototype. This is outlined in Section 6.2.1. Section 6.2.2 outlines research possibilities to extend the URDS architecture.

### 6.2.1 Extensions to the Prototype Implementation

Many extensions are possible of the current prototype. A few of these are:

- Discovery of components developed using other distributed computing models such as CORBA, Voyager, etc., by extending the native registries of the respective models in accordance with algorithms outlined in Section 4.8.1 of Chapter 4.

- A support for Federation of ICBs by providing for an implementation of the Link Manager Service in accordance with the algorithms outlined in Section 4.4.1 of Chapter 4.

- A support for Adapter Component Services. This will require further research in the area of developing adapter components by using glue and wrapper technology. The URDS architecture provides the Adapter Manager Service (refer Algorithms outlined in Section 4.5.1) for the discovery of these adapter components.

- A support for the failure detection capabilities in the Headhunter outlined in algorithm 4.6.1.6 of Chapter 4.

- The performance of the prototype can be optimized by fine tuning various parameters such as the time between periodic multicast announcements by the Headhunter, the time period between successive purge cycles to maintain the 'freshness' of the information returned as a result of a search, and the number of Headhunters present in the system per domain, etc.

- The performance of the prototype from an implementation perspective can be enhanced using alternative technologies for implementation. Section 5.5 in Chapter 5 outlines implementation strategies for enhancing the prototype.

- Testing the scalability and incorporating the feedback.

### 6.2.2. Enhancements of the URDS Architecture

Possible areas for enhancement of the current architecture are:

- *'Intelligent' Search Process:* The search propagation process in URDS can be made 'intelligent' i.e., the search process can use decision making capabilities to restrict the search space and perform more 'selective' searches. The URDS architecture proposed the use of query propagation policies, search scoping policies and function scoping policies to restrict the search space (See Section 4.3 of Chapter 4). The process can be further improved if the search process utilizes heuristics such as: a) past history and statistical results, and b) notion of acquaintances to narrow the search space. The QM can use heuristics to propagate a search to Headhunters and Link Managers. The QM can base the decision to propagate a query on parameters such as past history of availability, response time, number and quality of search results obtained from a Headhunter/Link Manager.

- *Mobile Headhunters:* The Headhunters in the current implementation use the network communication to gather information from the Active Registries. A possible enhancement could be to make the Headhunters 'mobile' i.e., the Headhunters move to the machines hosting the registries to gather information from the registries. The performance advantage of such an approach over the current approach needs to be carefully weighed out as it raises several security concerns.

- Usages of alternative schemes for techniques such as Security to create domain-specific (and restricted) versions of URDS are other directions of enhancement.

## 6.3 Summation

The thesis has presented an architecture and an implementation for the part of an infrastructure facilitating a semi-automatic construction of a distributed system that provides for the dynamic discovery of heterogeneous components and selection of components meeting the necessary requirements, including desired levels of QoS. The URDS architecture addresses essential design issues such as interoperability, QoS of software components, scalability, fault tolerance, security and network usage. These issues, considered in the design of URDS, make it unique as compared to other existing approaches. Interoperability is achieved by discovering components developed in several different component models. The discovery mechanism uses multicasting to detect native registries/lookup services of various component models that are extended to possess ‘active' and ‘introspective’ capabilities. The specifications capture the computational, functional, co-operational, auxiliary attributes and QoS metrics of their registered components. Flexibility in query formatting is achieved by providing support for natural language-like client requests. URDS is organized in a federated hierarchical structure as a scalability mechanism. Failure tolerance is handled through periodic announcements by failure prone entities and through information caching. Security is provided through authentication of the principals involved, access control to multicast address resources, and encryption of data transmitted. The URDS architecture coupled with the UA presents a promising solution for creating DCS by integrating geographically scattered, heterogeneous software components.

# APPENDIX

## Source Code

**umm.entity.beans.AuthenticatedPacket**

```
package umm.entity.beans;

import java.io.*;
import javax.crypto.*;
import java.security.*;

/**
 * The AuthenticatedPacket encapsulates the multicast address and
 * secret-key returned to the principals (Headhunters/ActiveRegistries) by the Domain
 * Security Manager.
 * Creation date: (03/01/2002 6:19:08 PM)
 * @author: Nanditha Nayani Siram
 */
public class AuthenticatedPacket implements Serializable {
	private SecretKey secretkey = null;
	private String mcastaddress = null;

/**
 * The AuthenticatedPacket Constructor
 */
public AuthenticatedPacket(SecretKey key, String address) {
	secretkey = key;
	mcastaddress = address;
}

/**
 * Returns Secret-Key
 */
public SecretKey getKey() {
	return secretkey;
}

/**
 * Returns Multicast Address
 */
public String getMCAddress() {
	return mcastaddress;
}

/**
 * Set Multicast Address
 */
public void setMCAddress(String address) {
	mcastaddress = address;
}

/**
```

```
 * set Secret-Key
 */
public void setKey(SecretKey key) {
	secretkey = key;
}
}
```

**umm.entity.beans.ComponentBean**

```
package umm.entity.beans;
import umm.helper.dataaccess.*;

import java.util.*;
import java.sql.*;
import java.io.*;

/**
 * The attributes of the ComponentBean class mirror the
 * fields of the table Component. The ComponentBean
 * internally holds a list of FunctionBean objects.
 * The ComponentBean has functionality built in to
 * persist it to a database.
 * Creation date: (11/14/2001 5:09:08 PM)
 * @author: Nanditha Nayani Siram
 */
public class ComponentBean implements Serializable
{
	private java.lang.String id = "";
	private java.lang.String name = "";
	private java.lang.String description = "";
	private java.lang.String function = "";
	private java.lang.String algorithm = "";
	private java.lang.String complexity = "";
	private java.lang.String technology = "";
	private java.lang.String preprocessingCollaborators = "";
	private double end2endDelay = 0;
	private double availability = 0;
	private java.lang.String mobility = "No";
	private java.util.Vector functionBeanList = new Vector();
	private java.lang.String status = "Active";
	private java.lang.String domain = "";

/**
 * ComponentBean constructor comment.
 */
public ComponentBean() {
	super();
}

/**
 * Add the function beans to list.
 */
public void addFunctionBeans(FunctionBean functionBean) {
	functionBeanList.addElement(functionBean);
}

/**
 * Populate attributes of bean from ResultSet.
 */
public void buildBean(ResultSet resultSet) throws java.lang.Exception {

	id = resultSet.getString("id");
	name = resultSet.getString("name");
	description = resultSet.getString("description");
	domain = resultSet.getString("domain");
	function = resultSet.getString("function");
	algorithm = resultSet.getString("algorithm");
	complexity = resultSet.getString("complexity");
	technology = resultSet.getString("technology");
```

```
        preprocessingCollaborators = resultSet.getString("collaborators");
        end2endDelay = resultSet.getDouble("end2endDelay");
        availability = resultSet.getDouble("availability");
        mobility = resultSet.getString("mobility");

}
/**
 * Return Algorithm
 */
public java.lang.String getAlgorithm() {
        return algorithm;
}
/**
 * Return availability
 */
public double getAvailability() {
        return availability;
}
/**
 * Return complexity
 */
public java.lang.String getComplexity() {
        return complexity;
}
/**
 * Return description
 */
public java.lang.String getDescription() {
        return description;
}
/**
 * Return domain
 */
public java.lang.String getDomain() {
        return domain;
}
/**
 * Return end2endDelay
 */
public double getEnd2endDelay() {
        return end2endDelay;
}
/**
 * Return function
 */
public java.lang.String getFunction() {
        return function;
}
/**
 * Return functionBeanList
 */
public java.util.Vector getFunctionBeanList() {
        return functionBeanList;
}
/**
 * Return id
 */
public java.lang.String getId() {
        return id;
}
/**
 * Return mobility
 */
public java.lang.String getMobility() {
        return mobility;
}
/**
 * Return name
 */
```

```
public java.lang.String getName() {
        return name;
}
/**
 * Return preprocessingCollaborators
 */
public java.lang.String getPreprocessingCollaborators() {
        return preprocessingCollaborators;
}
/**
 * Return status
 */
public java.lang.String getStatus() {
        return status;
}
/**
 * Return technology
 */
public java.lang.String getTechnology() {
        return technology;
}
/**
 * Set algorithm
 */
public void setAlgorithm(java.lang.String newAlgorithm) {
        algorithm = newAlgorithm;
}
/**
 * Set availability
 */
public void setAvailability(double newAvailability) {
        availability = newAvailability;
}
/**
 * Set complexity
 */
public void setComplexity(java.lang.String newComplexity) {
        complexity = newComplexity;
}
/**
 * Set description
 */
public void setDescription(java.lang.String newDescription) {
        description = newDescription;
}
/**
 * Set domain
 */
public void setDomain(java.lang.String newDomain) {
        domain = newDomain;
}
/**
 * Set end2endDelay
 */
 public void setEnd2endDelay(double newEnd2endDelay) {
        end2endDelay = newEnd2endDelay;
}
/**
 * Set function
 */
 public void setFunction(java.lang.String newFunction) {
        function = newFunction;
}
/**
 * Return functionBeanList
 */
public void setFunctionBeanList(java.util.Vector newFunctionBeanList) {
        functionBeanList = newFunctionBeanList;
}
```

```
/**
 * Set id
 */
public void setId(java.lang.String newId) {
	id = newId;
}
/**
 * Set mobility
 */
public void setMobility(java.lang.String newMobility) {
	mobility = newMobility;
}
/**
 * Set name
 */
public void setName(java.lang.String newName) {
	name = newName;
}
/**
 * Set preprocessingCollaborators
 */
public void setPreprocessingCollaborators(java.lang.String newPreprocessingCollaborators)
{
	preprocessingCollaborators = newPreprocessingCollaborators;
}
/**
 * Set status
 */
public void setStatus(java.lang.String newStatus) {
	status = newStatus;
}
/**
 * Set technology
 */
public void setTechnology(java.lang.String newTechnology) {
	technology = newTechnology;
}

/**
 * Persist Bean values to database.
 */
public void persistBean(SQLHelper sqlHelper) throws java.lang.Exception {

	String componentUpdateString =
	"INSERT INTO COMPONENT VALUES(" +
	 "'" + id + "'," +
	 "'" + name + "'," +
	 "'" + description + "'," +
	 "'" + function + "'," +
	 "'" + algorithm + "'," +
	 "'" + complexity + "'," +
	 "'" + domain + "'," +
	 "'" + technology + "'," +
	 "'" + preprocessingCollaborators + "'," +
	 end2endDelay + "," +
	 availability + "," +
	 "'" + mobility + "'" +
	 ")";

	 sqlHelper.updateTable(componentUpdateString);

	 for(int i=0;i<functionBeanList.size();i++)
	 {
	   FunctionBean functionBean = (FunctionBean) functionBeanList.get(i);
	   functionBean.persistBean(sqlHelper);
	 }

}
}//End of ComponentBean
```

**umm.entity.beans.FunctionBean**

```
package umm.entity.beans;
import umm.helper.dataaccess.*;

import java.util.*;
import java.io.*;

/**
 * The attributes of the FunctionBean mirror the fields
 * of the table Function. The FunctionBean has
 * functionality built in to persist it to a database.
 * Creation date: (11/15/2001 11:50:10 AM)
 * @author: Nanditha Nayani Siram
 */

public class FunctionBean implements Serializable{

	private java.lang.String functionName = "";
	private java.lang.String syntacticContract = "";
	private java.lang.String id = "";

/**
 * FunctionBean constructor comment.
 */
public FunctionBean() {
	super();
}
/**
 * Construct Bean from ResultSet.
 */
public void buildBean(java.sql.ResultSet resultSet) throws java.lang.Exception {

	id = resultSet.getString("id");
	functionName = resultSet.getString("function_name");
	syntacticContract = resultSet.getString("syntactic_contract");
}
/**
 * Return Function Name
 */
public java.lang.String getFunctionName() {
	return functionName;
}
/**
 * Return Sytactic Contract
 */
public java.lang.String getSyntacticContract() {
	return syntacticContract;
}
/**
 * Set Function Name
 */
public void setFunctionName(java.lang.String newFunctionName) {
	functionName = newFunctionName;
}
/**
 * Set Syntactic Contract
 */
public void setSyntacticContract(java.lang.String newSyntacticContract) {
	syntacticContract = newSyntacticContract;
}

/**
 * Return ID
 */
public java.lang.String getId() {
	return id;
}

/**
 * Set ID
 */
public void setId(java.lang.String newId) {
	id = newId;
}
/**
 * Persist Bean onto DB.
 */
public void persistBean(SQLHelper sqlHelper) throws Exception {

	String functionUpdateString =
	"INSERT INTO FUNCTION VALUES(" +
	"'" + id + "'," +
	"'" + functionName + "'," +
	"'" + syntacticContract + "')";

	 sqlHelper.updateTable(functionUpdateString);
}
}//end of FunctionBean
```

**umm.entity.beans.QueryBean**

```
package umm.entity.beans;

import java.util.*;
import java.io.*;

/**
 * The QueryBean encapsulates the attributes of a Query
 * received from the client. The bean also has the logic
 * associated with generating a SQL query based on the
 * attributes it holds.
 * Creation date: (11/15/2001 1:15:26 PM)
 * @author: Nanditha Nayani Siram
 */

public class QueryBean implements Serializable{
	private java.lang.String domain = "";
	private java.lang.String componentName = "";
	private java.lang.String componentDescription = "";
	private java.lang.String functionNames = "";
	private java.lang.String algorithms = "";
	private java.lang.String complexity = "";
	private java.lang.String technology = "";

	private java.lang.String availabilityConstraint = "";
	private boolean availabilityFlag = false;
	private double availabilityValue = 0;

	private boolean end2endDelayFlag = false;
	private java.lang.String end2endDelayConstraint = "";
	private double end2endDelayValue = 0;
	private java.lang.String mobility = "No";
	private int numOffers = 0;
	private int numMetrics = 0;
	private int hopcount;
	private java.lang.String requestID;

/**
 * QueryBean constructor comment.
 */
public QueryBean() {
	super();
}
/**
 * Returns algorithms Query
 */
```

```
public String getAlgorithmQuery() {

           String[] stringTokens = tokeniseString(algorithms);
               StringBuffer queryBuilder = new StringBuffer();
               queryBuilder.append(" ( ");

               for(int i =0;i<stringTokens.length;i++)
               {

                       String searchQuery =
                               "("+
                               " UPPER(ALGORITHM) LIKE '%" + stringTokens[i].toUpperCase()
+"%' " +
                               ")";

                       queryBuilder.append(searchQuery);

                       if((i+1)<stringTokens.length)
                         queryBuilder.append(" OR ");
               }

               queryBuilder.append(" ) ");
               return queryBuilder.toString();
}
/**
 * Return availability query.
 */
public String getAvailabilityQuery() {
               String availabilityQuery = null;

               if((availabilityConstraint != null) &&
!(availabilityConstraint.equalsIgnoreCase("None")))
               {
           availabilityQuery =
                         "(" +
                                 " AVAILABILITY " + availabilityConstraint + " " +
availabilityValue +
                         ")";

               }
               else
               {
                       availabilityQuery =
                         "(" +
                                " AVAILABILITY > 0" +
                         ")";
               }

          return availabilityQuery;
}
/**
 * Return complexity query.
 */
public String getComplexityQuery() {

               String[] stringTokens = tokeniseString(complexity);
               StringBuffer queryBuilder = new StringBuffer();
               queryBuilder.append(" ( ");

               for(int i =0;i<stringTokens.length;i++)
               {

                       String searchQuery =
                               "("+
                               " UPPER(COMPLEXITY) LIKE '%" +
stringTokens[i].toUpperCase() +"%' " +
                               ")";

                       queryBuilder.append(searchQuery);
                       if((i+1)<stringTokens.length)
                         queryBuilder.append(" OR ");
               }

               queryBuilder.append(" ) ");
               return queryBuilder.toString();
}
/**
 * Return component description query.
 */
public String getComponentDescriptionQuery() {

               String[] stringTokens = tokeniseString(componentDescription);
               StringBuffer queryBuilder = new StringBuffer();
               queryBuilder.append(" ( ");

               for(int i =0;i<stringTokens.length;i++)
               {

                       String searchQuery =
                               "("+
                               " UPPER(DESCRIPTION) LIKE '%" +
stringTokens[i].toUpperCase() +"%' " +
                               ")";

                       queryBuilder.append(searchQuery);

                       if((i+1)<stringTokens.length)
                         queryBuilder.append(" OR ");
               }

               queryBuilder.append(" ) ");
               return queryBuilder.toString();
}
/**
 * Return component name query.
 */
public String getComponentNameQuery() {

               String[] stringTokens = tokeniseString(componentName);
               StringBuffer queryBuilder = new StringBuffer();
               queryBuilder.append(" ( ");

               for(int i =0;i<stringTokens.length;i++)
               {

                       String searchQuery =
                               "("+
                               " UPPER(NAME) LIKE '%" + stringTokens[i].toUpperCase() +"%'
" +
                               ")";

                       queryBuilder.append(searchQuery);

                       if((i+1)<stringTokens.length)
                         queryBuilder.append(" OR ");
               }

               queryBuilder.append(" ) ");
               return queryBuilder.toString();
}
/**
 * Return Domain Query.
 */
public String getDomain() {
       return domain;
}
```

```
/**
 * Return End2EndDelay Query.
 */
public String getEnd2EndDelayQuery() {

               String end2endDelayQuery = null;

               if((end2endDelayConstraint != null) &&
!(end2endDelayConstraint.equalsIgnoreCase("None")))
               {
                       end2endDelayQuery =
                  "(" +
                   " END2ENDDELAY " + end2endDelayConstraint + " " + end2endDelayValue +
                   ")";
               }
               else
               {
                       end2endDelayQuery =
                   "(" +
                   " END2ENDDELAY > 0" +
                   ")";
               }

           return end2endDelayQuery;
}
/**
 * Return Function names Query.
 */
public String getFunctionNamesQuery() {

           String[] stringTokens = tokeniseString(functionNames);
           StringBuffer queryBuilder = new StringBuffer();

               queryBuilder.append(" ( ");
               for(int i =0;i<stringTokens.length;i++)
               {

                   String searchQuery =
                              "("+
                              " UPPER(function_name) LIKE '%" +
stringTokens[i].toUpperCase() +"%' " +
                              " OR " +
                              " UPPER(syntactic_contract) LIKE '%" +
stringTokens[i].toUpperCase() + "%' " +
                              ")";

                       queryBuilder.append(searchQuery);

                       if((i+1)<stringTokens.length)
                         queryBuilder.append(" OR ");
               }

               queryBuilder.append(" ) ");

        return queryBuilder.toString();
}
/**
 * Return Mobility Query.
 */
public String getMobilityQuery() {

               String mobilityQuery =
               "(" +
               " UPPER(MOBILITY) LIKE '%" + mobility.toUpperCase() + "%' "+
               ")";

           return mobilityQuery;
}
```

```
/**
 * Return Technology Query.
 */
public String getTechnologyQuery() {
       String technologyQuery =
              "(" +
              " UPPER(technology) LIKE '%" + technology.toUpperCase() + "%' "+
              ")";

       return technologyQuery;
}

/**
 * Return Hop Count.
 */
public int getHopcount() {
       return hopcount;
}

/**
 * Return requestID
 */
public java.lang.String getRequestID() {
       return requestID;
}

/**
 * Set Algorithms.
 */
public void setAlgorithms(java.lang.String newAlgorithms) {
       algorithms = newAlgorithms;
}
/**
 * Set availability Constraint.
 */
public void setAvailabilityConstraint(java.lang.String newAvailabilityConstraint) {
       availabilityConstraint = newAvailabilityConstraint;
}
/**
 * Set Availability Flag.
 */
public void setAvailabilityFlag(boolean newAvailabilityFlag) {
       availabilityFlag = newAvailabilityFlag;
}
/**
 * Set Avaliability Value.
 */
public void setAvailabilityValue(double newAvailabilityValue) {
       availabilityValue = newAvailabilityValue;
}
/**
 * Set Complexity.
 */
public void setComplexity(java.lang.String newComplexity) {
       complexity = newComplexity;
}
/**
 * Set Component Description.
 */
public void setComponentDescription(java.lang.String newComponentDescription) {
       componentDescription = newComponentDescription;
}
/**
 * Set Component Name.
 */
public void setComponentName(java.lang.String newComponentName) {
       componentName = newComponentName;
}
/**
```

```
 * Set Domain.
 */
public void setDomain(java.lang.String newDomain) {
       domain = newDomain;
}
/**
 * Set end2endDelayConstraint.
 */
public void setEnd2endDelayConstraint(java.lang.String newEnd2endDelayConstraint) {
       end2endDelayConstraint = newEnd2endDelayConstraint;
}
/**
 * Set end2EndDelayFlag.
 */
public void setEnd2endDelayFlag(boolean newEnd2endDelayFlag) {
       end2endDelayFlag = newEnd2endDelayFlag;
}
/**
 * Set end2endDelayValue.
 */
public void setEnd2endDelayValue(double newEnd2endDelayValue) {
       end2endDelayValue = newEnd2endDelayValue;
}
/**
 * Set function names.
 */
public void setFunctionNames(java.lang.String newFunctionNames) {
       functionNames = newFunctionNames;
}
/**
 * Set mobility.
 */
public void setMobility(java.lang.String newMobility) {
       mobility = newMobility;
}
/**
 * Set numMetrics.
 */
public void setNumMetrics(int newNumMetrics) {
       numMetrics = newNumMetrics;
}
/**
 * Set numOffers.
 */
public void setNumOffers(int newNumOffers) {
       numOffers = newNumOffers;
}
/**
 * Set Technology
 */
public void setTechnology(java.lang.String newTechnology) {
       technology = newTechnology;
}

/**
 * Set hopCount
 */
public void setHopcount(int newHopcount) {
       hopcount = newHopcount;
}
/**
 * Set requestID
 */
public void setRequestID(java.lang.String newRequestID) {
       requestID = newRequestID;
}

/**
 * Tokenize keywords.
 */
public String[] tokeniseString(String keyWords) {

        String[] stringTokens = null;

         if (keyWords != null) {
                        StringTokenizer strTok = new StringTokenizer(keyWords);
                        int numTokens = strTok.countTokens();
                        stringTokens = new String[numTokens];
                        int i = 0;
                        while (strTok.hasMoreTokens()) {
                                stringTokens[i] = strTok.nextToken();
                                i++;
                        }
        }

         return stringTokens;

}
/**
 * Return resultant composed query.
 */
public String getQuery() {

        String componentTable = "COMPONENT A";
        String functionTable = "FUNCTION B";

        String baseQuery = "SELECT * FROM " + componentTable + ", " +
                                    functionTable +" WHERE (A.ID = B.ID)";

        String bodyQuery = "";

        if ((componentName !=null) && (!componentName.equals(""))) {
                bodyQuery = bodyQuery + " AND " + getComponentNameQuery();

        }
        if ((componentDescription !=null) && (!componentDescription.equals(""))) {
                bodyQuery = bodyQuery + " AND " + getComponentDescriptionQuery();
        }
        if ((functionNames !=null) && (!functionNames.equals(""))) {
                bodyQuery = bodyQuery + " AND " + getFunctionNamesQuery();
        }

        if ((algorithms !=null) && (!algorithms.equals(""))) {
                bodyQuery = bodyQuery + " AND " + getAlgorithmQuery();
        }

   if ((complexity !=null) && (!complexity.equals(""))) {
                bodyQuery = bodyQuery + " AND " + getComplexityQuery();
        }

   if ((technology !=null) && (!technology.equals(""))) {
                bodyQuery = bodyQuery + " AND " + getTechnologyQuery();
        }

   if (availabilityFlag) {
                bodyQuery = bodyQuery + " AND " + getAvailabilityQuery();
        }

   if (end2endDelayFlag) {
                bodyQuery = bodyQuery + " AND " + getEnd2EndDelayQuery();
        }

   if ((mobility !=null) && (!mobility.equalsIgnoreCase("No"))) {
                bodyQuery = bodyQuery + " AND " + getMobilityQuery();
        }

        String query = baseQuery + bodyQuery;
        return query;
}

}//end QueryBean
```

**umm.helper.dataaccess.DSMRepositoryHelper**

```
package umm.helper.dataaccess;

import java.sql.*;
import java.util.*;

/**
 * This class performs functions associated with
 * accessing the DSM_Repository to retrieve
 * user-domain mappings and for user authentication.
 * Creation date: (11/15/2001 1:15:26 PM)
 * @author: Nanditha Nayani Siram
 */

public class DSMRepositoryHelper {

private static SQLHelper sqlHelper = null;

/**
 * Initialize SQLHelper
 */
public static void initialize() {
	try {
		sqlHelper = new SQLHelper();

	} catch (Exception e) {
		System.out.println(e);
	}

}

/**
 * authenticate principal against DB.
 */
public static boolean authenticateUser(
	String sUserType,
	String sUserName,
	String sPassword) {
	boolean isAuthenticated = false;

	try {
		String sUserQuery =
			"SELECT UserName From Users "
				+ "WHERE ( ( Users.UserName = '"
				+ sUserName
				+ "') AND ( Users.Password = '"
				+ sPassword
				+ "' ) AND ( Users.UserType = '"
				+ sUserType
				+ "' ) )";

		// execute the Query
		ResultSet resultSet = sqlHelper.executeQuery(sUserQuery);

		if (!resultSet.next()) {
			// the database has no results - therefore the user is not
authenticated
			return false;
		}

		// retrieve the resultset
		String sResult = resultSet.getString(1);

		// if the username is returned,
                if (sResult != null) {
                        // the user has been successfully authenticated
                        isAuthenticated = sUserName.equals(sResult);
                }

        } catch (SQLException sqlE) {
                sqlE.printStackTrace();
                return false;
        } catch (Exception e) {
                e.printStackTrace();
                return false;
        }

        return isAuthenticated;
}

/**
 * Build hashtable from resultset.
 */
private static Hashtable getFeaturesFromResultSet(ResultSet resultSet)
        throws SQLException {

        Hashtable hFeatures = new Hashtable();

        String sFeatureName = null;
        String sFeatureValue = null;

        while (resultSet.next()) {
                sFeatureName = resultSet.getString(1);
                sFeatureValue = resultSet.getString(2);
                hFeatures.put(sFeatureName, sFeatureValue);
        }

        return hFeatures;
}

/**
 * Load from DomainList and Permissions tables.
 */
public static Hashtable loadDomainList() {
        Hashtable hFeatures = null;

        String sDomainListQuery =
                "SELECT DomainList.DomainAddress, Permissions.PermissionName From
DomainList, Permissions  "
                        + " WHERE (DomainList.DomainID = Permissions.PermissionID)";

        try {
                // execute the Query
                ResultSet resultSet = sqlHelper.executeQuery(sDomainListQuery);
                hFeatures = getFeaturesFromResultSet(resultSet);

        } catch (SQLException sqlE) {
                sqlE.printStackTrace();
                return null;
        } catch (Exception e) {
                e.printStackTrace();
                return null;
        }

        return hFeatures;
}

/**
 * Load from Users and Permissions.
 */
public static Hashtable loadUserDomainMapping() {
        Hashtable hFeatures = null;
        String sUserDomainQuery =
                "SELECT Users.UserName, Permissions.PermissionName "
                        + "FROM Users, Permissions, User_Permission_Xref "
                        + " WHERE ( "
                        + "(User_Permission_Xref.PermissionID = Permissions.PermissionID)
AND "
                        + "(User_Permission_Xref.UserID = Users.UserID ) )";

        try {
                // execute the Query
                ResultSet resultSet = sqlHelper.executeQuery(sUserDomainQuery);
                hFeatures = getFeaturesFromResultSet(resultSet);

        } catch (SQLException sqlE) {
                sqlE.printStackTrace();
                return null;
        } catch (Exception e) {
                e.printStackTrace();
                return null;
        }

        return hFeatures;
}

/**
 * Load from Permissions table.
 */
public static ArrayList getListOfDomains() {

        String sListofDomainsQuery = "SELECT PermissionName From Permissions";

        try {
                // execute the Query
                ResultSet resultSet = sqlHelper.executeQuery(sListofDomainsQuery);
                // position to first record
                boolean moreRecords = resultSet.next();
                // If there are no records, display a message
                if (!moreRecords) {
                        return null;
                } else {
                        ArrayList listOfDomains = new ArrayList();
                        do {
                                String domain = resultSet.getString("PermissionName");
                                listOfDomains.add(domain);
                        } while (resultSet.next());
                        return listOfDomains;
                } //end else

        } catch (SQLException sqlE) {
                sqlE.printStackTrace();
                return null;
        } catch (Exception e) {
                e.printStackTrace();
                return null;
        }
}
}//end of DSMRepositoryHelper
```

**umm.helper.dataaccess.MetaRepositoryHelper**

```
package umm.helper.dataaccess;

import umm.entity.beans.*;

import java.sql.*;
import java.util.*;

/**
 * This class performs functions associated with accessing
 * the Meta_Repository to retrieve search results.
 * Creation date: (11/15/2001 1:15:26 PM)
 * @author: Nanditha Nayani Siram
 */
public class MetaRepositoryHelper {

private java.util.Hashtable resultTable;
private QueryBean queryBean;

/**
 * Constructor 1
 */
public MetaRepositoryHelper() {
	super();
}

/**
 * Constructor 2
 */
public MetaRepositoryHelper(QueryBean newQueryBean) {
	queryBean = newQueryBean;
}

/**
 * Excecute Query against MR and return search results.
 */
public Hashtable getSearchResultTable() throws java.lang.Exception {

	SQLHelper sqlHelper = new SQLHelper();

	String searchQuery = queryBean.getQuery();

	if (searchQuery == null || searchQuery.equals(""))
		throw new Exception("No Parameters Passed For Search");

	ResultSet resultSet = sqlHelper.executeQuery(searchQuery);

	// position to first record
	boolean moreRecords = resultSet.next();
	// If there are no records, display a message
	if (!moreRecords) {
		sqlHelper.shutDown();
		throw new Exception("No Records Matching Search Criteria");
	} else {
		resultTable = new Hashtable();

		ComponentBean componentBean = null;

		// get row data
		do {

			String ID = resultSet.getString("id");

			if (!resultTable.isEmpty() && resultTable.containsKey(ID)) {
				componentBean = (ComponentBean) resultTable.get(ID);
				FunctionBean functionBean = new FunctionBean();
				functionBean.buildBean(resultSet);
				componentBean.addFunctionBeans(functionBean);

			} else {

				componentBean = new ComponentBean();

				componentBean.buildBean(resultSet);

				FunctionBean functionBean = new FunctionBean();
				functionBean.buildBean(resultSet);

				componentBean.addFunctionBeans(functionBean);
```

```
                              resultTable.put(ID, componentBean);

                        }

                } while (resultSet.next());

        } //end of else

        sqlHelper.shutDown();

        return resultTable;
}
}//end of MRHelper
```

**umm.helper.dataaccess.SQLHelper**

```
package umm.helper.dataaccess;

import java.util.*;
import java.sql.*;

/**
 * This is the class which serves as a connection to the
 * oracle database. It establises the database connection
 * and executes queries which either select/update the
 * tables of the database as well as execute stored
 * procedures.
 * Creation date: (9/14/2001 12:57:07 PM)
 * @author: Nanditha Nayani Siram
 */
public class SQLHelper {
        private java.sql.Connection dbconn = null;
        private java.sql.Statement statement = null;

/**
 * Constructor
 */
public SQLHelper() throws java.lang.Exception {

        //-------------------------------------------------------------
        // Get Connection to database.
        // The URL specifying the database to which
        // this program connects using JDBC
        //-------------------------------------------------------------

        String url = "jdbc:oracle:thin:@phoenix.cs.iupui.edu:1521:OS80";

        String username = "nnayani";
        String password = "nnayani";

        // Load the driver to allow connection to the database
        try {
                Class.forName("oracle.jdbc.driver.OracleDriver");

                dbconn = DriverManager.getConnection(url, username, password);

                statement = dbconn.createStatement();

        } catch (ClassNotFoundException cnfex) {

                System.err.println("Failed to load driver.");
                cnfex.printStackTrace();
                System.exit(1); // terminate program

        } catch (SQLException sqlex) {

                System.err.println("\n Unable to connect to Oracle Server");
                sqlex.printStackTrace();
                System.exit(1); // terminate program
        }
```

```
}

/**
 * Commit Transaction
 */
public final void commitTransaction() throws java.lang.Exception {
        try {
                dbconn.commit();
        } catch (SQLException sqle) {
                throw new Exception(
                        "\n SQL Exception during commitTransaction with message :" +
sqle.getMessage());
        }
}

/**
 * Execute a Query
 */
public ResultSet executeQuery(String query) throws java.lang.Exception {

        ResultSet resultSet = null;

        try {
                resultSet = statement.executeQuery(query);
        } catch (SQLException sqle) {
                throw new Exception(
                        "\n SQL Exception during executeQuery with message:" +
sqle.getMessage());
        }

        return resultSet;

}

/**
 * Return DB Connection
 */
public java.sql.Connection getDbconn() {
        return dbconn;
}

/**
 * Return Statement
 */
public java.sql.Statement getStatement() {
        return statement;
}
/**
 * Turn off Auto Commit before initiating transaction
 */
public final void initiateTransaction() throws java.lang.Exception {
        try {
                dbconn.setAutoCommit(false);
        } catch (SQLException sqle) {
                throw new Exception(
                        "\n SQL Exception during initiateTransaction with message :"
                                + sqle.getMessage());
        }
}
/**
 * Perform Rollback
 */
public void rollbackTransaction() throws java.lang.Exception {
        try {
                dbconn.rollback();
        } catch (SQLException sqle) {
                throw new Exception(
                        "\n SQL Exception during rollbackTransaction with message :"
```

```
                              + sqle.getMessage());
        }
}
/**
 * Set DB Connection
 */
public void setDbconn(java.sql.Connection newDbconn) {
        dbconn = newDbconn;
}
/**
 * Set Statement
 */
public void setStatement(java.sql.Statement newStatement) {
        statement = newStatement;
}
/**
 * Close connection
 */
public void shutDown() throws java.lang.Exception {

        try {

                if (statement != null)
                        statement.close();

                if (dbconn != null)
                        dbconn.close();

        } catch (Exception e) {
                throw new Exception(e.getMessage());
        }

}
/**
 * Update DB Table
 */
public void updateTable(String updateString) throws java.lang.Exception {

        try {
                dbconn.setAutoCommit(false);
                statement.executeUpdate(updateString);
                dbconn.commit();
        } catch (SQLException sqle) {
                throw new Exception(
                        "\n SQL Exception from function updateTable with message:" +
sqle.getMessage());
        }

}
}
```

**umm.helper.dependent.CreateKey**

```
package umm.helper.dependent;

import javax.crypto.*;
import java.io.*;
import java.security.*;

/**
 * This utility class is used to generate secret-keys.
 * Creation date: (10/14/2001 5:09:08 PM)
 * @author: Nanditha Nayani
 */

public class CreateKey
{
/**
 * Constructor
 */
public CreateKey() {
}
/**
 * generate and Return Secret-Key.
 */
public static SecretKey getKey() {
        SecretKey desKey = null;
        try {
                Provider sunJce = new com.sun.crypto.provider.SunJCE();
                Security.addProvider(sunJce);

                KeyGenerator keyGen = KeyGenerator.getInstance("DES");
                desKey = keyGen.generateKey();

        } catch (Exception e) {
                System.out.println(e);
        }
        return desKey;
}
}
```

**umm.helper.dependent.CryptObj**

```
package umm.helper.dependent;

import javax.crypto.*;
import java.security.*;

/**
 * This class provides the utility functions
 * for encrypting and decrypting data.
 * Creation date: (10/14/2001 5:09:08 PM)
 * @author: Nanditha Nayani
 */

public class CryptObj {

        private Cipher encryptCipher = null;
        private Cipher decryptCipher = null;

   /**
    * Constructor
    */
   public CryptObj(SecretKey desKey)
   {

        try
        {
                Provider sunJce = new com.sun.crypto.provider.SunJCE();
                Security.addProvider(sunJce);

                // get cipher object for encryption
                encryptCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
                encryptCipher.init(Cipher.ENCRYPT_MODE, desKey);

                decryptCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
                decryptCipher.init(Cipher.DECRYPT_MODE, desKey);
        }
        catch (Exception e)
        {
          System.out.println(e);
        }

   }

   /**
   * Encrypt Object
   */
   public Object enCryptObj(java.io.Serializable inObj)
   {
          SealedObject sealObj = null;

          try{

                sealObj = new SealedObject(inObj,encryptCipher);
          }
          catch(Exception e)
          {
                System.out.println(e);
          }
          return sealObj;

   }
  /**
   * Decrypt Object
   */
   public Object deCryptObj(SealedObject inObj)
   {
          Object thisObj = null;

          try{

                thisObj = inObj.getObject(decryptCipher);
          }
          catch(Exception e)
          {
                System.out.println(e);
          }

          return thisObj;

   }
 }//end of CryptObj
```

**umm.helper.dependent. MulticastReceiver**

```
package umm.helper.dependent;

import umm.helper.dependent.*;
import umm.entity.beans.*;
import umm.services.*;

import java.net.*;
import java.io.*;
import java.util.*;
import java.rmi.*;

import java.security.*;
import javax.crypto.*;

/**
 * This class operates on a Thread, which constantly listens
 * for multicast messages on a MulticastSocket and receives
 * the incoming DatagramPackets. This thread utilizes the
 * CryptObject and ObjectSerializer Helper objects to decrypt
 * and reconstruct the multicast messages received.
 * Creation date: (10/15/2001 11:50:10 AM)
 * @author: Nanditha Nayani Siram
 */

public class MulticastReceiver implements Runnable {

       private InetAddress groupAddress = null;
       private MulticastSocket multicastSocket = null;
       private int port = 10000;
       private CryptObj cryptObject = null;
       private javax.crypto.SealedObject encryptedRegistryLocation = null;
       private ObjectSerializer objSerializer = null;
       private java.lang.String registryLocation = null;
```

```
public void run() {
        try {
                multicastSocket = new MulticastSocket(port);
                multicastSocket.joinGroup(groupAddress);
                System.out.println("\n Active Registry joined multicast group at : " +
groupAddress);

                while (true) {
                        byte[] buffer = new byte[1024];
                        //Receiving data
                        DatagramPacket dataPacket = new DatagramPacket(buffer,
buffer.length);
                        multicastSocket.receive(dataPacket);

                         //----        Without Encryption -------------
                         // String dataString = new String(dataPacket.getData());

                         //----         With Encryption  --------------

            byte[] encryptedHHLoc = dataPacket.getData();
                        objSerializer.setBytes(encryptedHHLoc);
                        SealedObject encryptedObject = (SealedObject)
objSerializer.getObject();
                        String dataString = (String)
cryptObject.deCryptObj(encryptedObject);
            System.out.println("Active Registry Received Encrypted Multicasted Headhunter
Location : " + dataString);
                        try {

                                System.setSecurityManager(new RMISecurityManager());
                                IHeadhunter hhunter = (IHeadhunter)
Naming.lookup(dataString.trim());

                                //Unicast information to headhunter
                                hhunter.receiveUnicastCommunication(registryLocation);

                                System.out.println( "Active Registry Unicast to Headhunter
" +
                                        dataString + " its Location " + registryLocation);


                        } catch (Exception e) {
                                System.out.println(e);

                        }
                } //end of while

        } catch (IOException ioe) {
                System.out.println(ioe);

        } //end of try-catch
        finally {
                if (multicastSocket != null) {
                        try {
                                multicastSocket.leaveGroup(groupAddress);
                                multicastSocket.close();
                        } catch (IOException ioe) {
                                System.out.println(ioe);

                        } //end of try-catch

                } //end of if
        } //end of finally

} //end of run

//Constructor
public MulticastReceiver(
        int mcastPort,
        AuthenticatedPacket authPacket,
        String regLocation) {

        try {
                groupAddress = InetAddress.getByName(authPacket.getMCAddress());
                port = mcastPort;
                cryptObject = new CryptObj(authPacket.getKey());
                objSerializer = new ObjectSerializer();
                registryLocation = regLocation;
                encryptedRegistryLocation = (SealedObject)
cryptObject.enCryptObj(registryLocation);

        } catch (Exception e) {
                System.out.println(e);
                System.exit(1);

        } //end of try-catch

} //end of Receiver
} //end of class MulticastReceiver
```

**umm.helper.dependent. MulticastSender**

```
package umm.helper.dependent;

import umm.helper.dependent.*;
import umm.entity.beans.*;

import java.net.*;
import java.security.*;
import javax.crypto.*;

/**
 * This class operates as a Thread which executes periodically.
 * This Thread has a connection to a MulticastSocket to which it
 * keeps multicasting DatagramPackets at regular intervals of time.
 * This thread utilizes the CryptObject and ObjectSerializer Helper
 * objects to multicast encrypted serialized messages.
 * Creation date: (10/15/2001 11:50:10 AM)
 * @author: Nanditha Nayani Siram
 */

public class MulticastSender implements Runnable{

        private InetAddress inetAddress = null;
        private DatagramPacket dataPacket = null;
        private int port = 10000;
        private byte ttl = (byte) 1;
        private String hostLocation = "";
        private byte[] buffer;
        private MulticastSocket multicastSocket = null;
        private long TPmcast = 0;
        private CryptObj cryptObject = null;
    private ObjectSerializer objSerializer = null;

public void run()
{
        Thread CurrentThread = Thread.currentThread();

           try
           {
                 while(true)
                 {
                dataPacket =
                        new DatagramPacket(buffer,buffer.length,inetAddress,port);
                       multicastSocket.send(dataPacket,ttl);
                       System.out.println("\n Multicasting Encrypted HH Location : " +
hostLocation);
                       CurrentThread.sleep(TPmcast);
```

```
                  }//end of while

                }
                catch(InterruptedException ie){

                        System.out.println(ie.getMessage());

            }//end of try-catch
                catch(SocketException se)
                {
                       System.out.println(se);
                }
                catch(Exception e)
                {
                         System.out.println(e);
                }//end of try-catch

}//end of run

    public MulticastSender(
    long mcastTime,
       int mcastPort,
       AuthenticatedPacket authPacket,
       String headHunterLoc) {

       try {

               TPmcast = mcastTime;
               inetAddress = InetAddress.getByName(authPacket.getMCAddress());
               port = mcastPort;
               hostLocation = headHunterLoc;
               cryptObject = new CryptObj(authPacket.getKey());


                //----- Without Encryption -----------
                // buffer = hostLocation.getBytes();

                //----   With Encryption -----------------
               //Create the encrypted host location by sealing
               //location with secret-key and serializing this object for transmission

        SealedObject sealedHostLocation = (SealedObject)
cryptObject.enCryptObj(hostLocation);
        objSerializer = new ObjectSerializer();
               objSerializer.setObject(sealedHostLocation);
               buffer = objSerializer.getBytes();


               multicastSocket = new MulticastSocket();
               multicastSocket.joinGroup(inetAddress);
        System.out.println("Headhunter joined multicast group:" + inetAddress);

       } catch (SocketException se) {
               System.out.println(se);
       } catch (Exception e) {
               System.out.println(e);
               System.exit(1);
       }

}}//end of class MulticastSender
```

**umm.helper.dependent.ObjectSerializer**

```
package umm.helper.dependent;

import java.io.*;

/**
 * This ObjectSerializer class is a mechanism to
 * transmit Java objects over the network.
 * It makes use of the Java serialization mechanism.
 * Creation date: (10/14/2001 5:09:08 PM)
 * @author: Nanditha Nayani Siram
 */

public class ObjectSerializer{
        Object myObj;

        public ObjectSerializer() {
        super();
        }

        public ObjectSerializer(Object o) {
        setObject(o);
        }

        /**
        Interprets the bytes as a serialized object
        */
        public ObjectSerializer(byte[] b) {
        setBytes(b);
        }

        /**
        Get the contained Object
        */
        public Object getObject() {
        return myObj;
        }

        /**
        Set the contained Object
        */
        public void setObject(Object o) {
        myObj = o;
        }

        /**
        Serializes the contained object into a byte array
        */
        public byte[] getBytes() {
        try {
            ByteArrayOutputStream ostream = new ByteArrayOutputStream(256);
                        ObjectOutputStream p = new ObjectOutputStream(ostream);
                        p.writeObject(myObj);
            p.flush();
                        byte[] rtnVal = ostream.toByteArray();
                        ostream.close();
            return rtnVal;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        }

        /**
        Interprets the bytes as a serialized object and sets the contained reference
        to the unserialized version of the serialized object
        */
        public void setBytes(byte[] b) {
        try {
            ByteArrayInputStream is = new ByteArrayInputStream(b);
            ObjectInputStream p = new ObjectInputStream(is);
            myObj = p.readObject();
            is.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        }

}// End class ObjectSerializer
```

**umm.helper.dependent. RequestProcessor**

```
package umm.helper.dependent;

import umm.entity.beans.*;
import umm.services.*;

import java.util.*;
import java.rmi.*;


/**
 * RequestProcessor provides the glue in the Web tier
 * for holding the application components together.
 * It contains logic that needs to be executed for each
 * request. RequestProcessor collaborates with the
 * QueryManager and gets a Hashtable of search results
 * from the QueryManager which it forwards to the View
 * ComponentList.jsp.
 * Creation date: (8/16/2001 1:08:35 PM)
 * @author: Nanditha Nayani
 */
public class RequestProcessor {

  private java.util.Hashtable resultTable = null;
  private IQueryManager qm = null;
  private QueryBean queryBean = null;


/**
 * RequestProcessor constructor comment.
 */
public RequestProcessor() throws Exception {

        String qmLocation = "//magellan.cs.iupui.edu:8500/QueryManager";
        qm = (IQueryManager) Naming.lookup(qmLocation);
         System.out.println("Looked up QM");

}

public void clearResults() {
        resultTable = null;
}

public java.util.Hashtable getResultTable() throws Exception {

        if (resultTable == null) {
                resultTable = qm.getSearchResultTable(queryBean);
        }

        return resultTable;
}

public void setResultTable(java.util.Hashtable newResultTable) {
        resultTable = newResultTable;
}

public QueryBean getQueryBean() {
        return queryBean;
}
public void setQueryBean(QueryBean newQueryBean) {
        queryBean = newQueryBean;
        resultTable = null;
}
}
```

**umm.helper.dependent.UniFrameIntrospector**

```
package umm.helper.dependent;

import javax.servlet.http.*;
import java.lang.reflect.*;
import java.beans.*;
import java.util.*;

/**
 * This utility class uses reflection to analyze the properties
 * of an object and retrieve the value corresponding to a
 * specific attribute.
 * Creation date: (6/11/2001 9:18:51 AM)
 * @author: Nanditha Nayani Siram
 */
public class UniFrameIntrospector {

public UniFrameIntrospector() {
	super();
}

/**
* Introspect Bean and return Property
*/
public static Object getProperty(Object bean, String propertyName)
	throws UniFrameIntrospectorException {

	Object property = null;
	Method method = null;
	Object[] args = null;

	try {

		BeanInfo info = Introspector.getBeanInfo(bean.getClass());
		PropertyDescriptor[] pds = info.getPropertyDescriptors();

		for (int i = 0; pds != null && i < pds.length; i++) {
			if (pds[i].getName().equals(propertyName)) {
				method = pds[i].getReadMethod();
				break;
			}
		}
	} catch (IntrospectionException e) {
		throw new UniFrameIntrospectorException(
			"Error analyzing the bean class: " + e.getMessage());
	}

	if (method == null)
		throw new UniFrameIntrospectorException(
			"Property " + propertyName + " not found");

	try {

		property = method.invoke(bean, args);
		System.out.println("Active Registry obtained UMM Spec URL by Introspection
: " + property);

	} catch (Exception e) {
		throw new UniFrameIntrospectorException(
			e.getClass().getName()
				+ ": "
				+ "Failed to get property "
				+ propertyName
				+ ", message: "
				+ e.getMessage());
	}

	return property;
}
}
```

**umm.helper.dependent.UniFrameIntrospectorException**

```
package umm.helper.dependent;

/**
 * Creation date: (10/5/2001 12:55:32 PM)
 * @author: Nanditha Nayani Siram
 */
public class UniFrameIntrospectorException extends Exception {
public UniFrameIntrospectorException() {
        super();
}
public UniFrameIntrospectorException(String s) {
        super(s);
}
}
```

**umm.helper.dependent.UniFrameSpecificationParser**

```
package umm.helper.dependent;
import umm.entity.beans.*;


import java.io.IOException;
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.apache.xerces.parsers.DOMParser;
import java.util.*;

/**
 * This class is used to parse a UniFrame XML
 * specification file and construct an instance
 * of the ComponentBean.
 * Creation date: (9/14/2001 12:57:07 PM)
 * @author: Nanditha Nayani Siram
 */

public class UniFrameSpecificationParser {

  private ComponentBean cbean = new ComponentBean();
  private Vector functionVector = new Vector();
  private boolean populatedFlag = false;
  private Vector syntaxVector = new Vector();

/**
 * Constructor
 */
public UniFrameSpecificationParser(String uri) {

        // Instantiate your vendor's DOM parser implementation
        DOMParser parser = new DOMParser();
        try {
                parser.parse(uri);
                Document doc = parser.getDocument();

                // Read the document from the DOM tree.
                readNode(doc);
        } catch (IOException e) {
                System.out.println("Error reading URI: " + e.getMessage());
        } catch (Exception ex) {
                System.out.println("Error in parsing: " + ex.getMessage());
        }

}

/**
```

```
 * Return Component Bean.
 */
public ComponentBean getComponentBean() {

        if (populatedFlag == false) {
                for (int i = 0; i < functionVector.size(); i++) {
                        FunctionBean fBean = new FunctionBean();
                        fBean.setId(cbean.getId());
                        fBean.setFunctionName((String) functionVector.get(i));
                        fBean.setSyntacticContract((String) syntaxVector.get(i));
                        cbean.addFunctionBeans(fBean);
                }
                populatedFlag = true;
        }
        return cbean;
}

/**
 * Parse through XML tree using recursion.
 */
private void readNode(Node node) {

        if (node.getNodeName().equalsIgnoreCase("ComponentName")) {
                cbean.setName(node.getFirstChild().getNodeValue());
        } else
                if (node.getNodeName().equalsIgnoreCase("Description")) {
                        cbean.setDescription(node.getFirstChild().getNodeValue());
                } else
                        if (node.getNodeName().equalsIgnoreCase("Function")) {
                                functionVector.add(node.getFirstChild().getNodeValue());
                        } else
                                if (node.getNodeName().equalsIgnoreCase("ID")) {
                                        cbean.setId(node.getFirstChild().getNodeValue());
                                } else
                                        if (node.getNodeName().equalsIgnoreCase("Purpose")) {

        cbean.setFunction(node.getFirstChild().getNodeValue());
                                        } else
                                                if
(node.getNodeName().equalsIgnoreCase("Algorithm")) {

        cbean.setAlgorithm(node.getFirstChild().getNodeValue());
                                                } else
                                                        if
(node.getNodeName().equalsIgnoreCase("Complexity")) {

        cbean.setComplexity(node.getFirstChild().getNodeValue());
                                                        } else
                                                                if
(node.getNodeName().equalsIgnoreCase("Contract")) {

        syntaxVector.add(node.getFirstChild().getNodeValue());
                                                                } else
                                                                        if
(node.getNodeName().equalsIgnoreCase("Technology")) {

        cbean.setTechnology(node.getFirstChild().getNodeValue());
                                                                        } else
                                                                                if
(node.getNodeName().equalsIgnoreCase("PreprocessingCollaborators")) {

        cbean.setPreprocessingCollaborators(node.getFirstChild().getNodeValue());
                                                                                } else
                                                                                        if
(node.getNodeName().equalsIgnoreCase("Mobility")) {

        cbean.setMobility(node.getFirstChild().getNodeValue());
                                                                                        } else
        if (node.getNodeName().equalsIgnoreCase("Availability")) {

        cbean.setAvailability(

                (new Double(node.getFirstChild().getNodeValue()).doubleValue()));

        } else

        if (node.getNodeName().equalsIgnoreCase("End2EndDelay")) {

                cbean.setEnd2endDelay(

                        (new Double(node.getFirstChild().getNodeValue()).doubleValue()));

        }

        // recurse on each child
        NodeList children = node.getChildNodes();
        if (children != null) {
                for (int i = 0; i < children.getLength(); i++) {
                        readNode(children.item(i));
                }
        }
}
}
```

**umm.services.ActiveRegistry**

```
package umm.services;

import umm.entity.beans.*;
import umm.helper.dependent.*;
import umm.services.*;


import java.rmi.registry.*;
import java.rmi.*;
import java.net.*;
import java.util.*;
import java.rmi.server.*;

/**
 * The ActiveRegistry class receives multicast messages from the Headhunter
 * by using the MulticastReceiver class. Interaction with the Headhunter for
 * passing its contact information and component information are done through
 * RMI-JRMP. It also communicates with the DomainSecurityManager for
 * authentication purposes using RMI-JRMP. The ActiveRegistry uses the
 * UniFrameIntrospector for the purpose of obtaining the UniFrame specifications
 * of the components registered with it. It also uses the UniFrameSpecificationParser
 * for parsing the specifications and building instances of ComponentBean objects,
 * which it returns to the Headhunter.
 * Creation date: (11/14/2001 5:09:08 PM)
 * @author: Nanditha Nayani Siram
 */

public class ActiveRegistry extends UnicastRemoteObject implements IActiveRegistry
{

    private int port = 9000;
        private String userType = "Registry";

/**
 * Obtain URL List of Registered Components.
 */
private String[] list() {

        String[] listOfURLS = null;
        try {
                listOfURLS = Naming.list("//magellan.cs.iupui.edu:9000");
        } catch (java.rmi.RemoteException e) {
                System.out.println(e.getMessage());
        } catch (Exception e) {
                System.out.println(e.getMessage());
        }

        return listOfURLS;
}

public static void main(String[] args) {

        int rmiRegistryPort = 9000;
        int mcastPort = 10000;
        String userName = "Reg1";
        String password = "Reg1";
        String domain = "Finance";
        String activeRegistryLocation = "//magellan.cs.iupui.edu:8500/ActiveRegistry";
        String dsmLocation = "//magellan.cs.iupui.edu:8500/DomainSecurityManager";

        try {
                System.setSecurityManager(new RMISecurityManager());
                Naming.rebind(
                        activeRegistryLocation,
                        new ActiveRegistry(
                                rmiRegistryPort,
                                mcastPort,
                                userName,
                                password,
                                domain,
                                activeRegistryLocation,
                                dsmLocation)
                        );
                System.out.println("ActiveRegistry is ready.");
        } catch (Exception e) {
                System.out.println("ActiveRegistry failed: " + e);
        }
}

/**
 * The ActiveRegistry Constructor
 */
public ActiveRegistry(
        int rmiRegistryPort,
        int mcastPort,
        String userName,
        String password,
        String domain,
        String activeRegistryLocation,
        String dsmLocation)
        throws RemoteException {

        try {

                String rmiRegistryLocation = (InetAddress.getLocalHost()).toString();
                port = rmiRegistryPort;

                //Create the registry on this host and port
                LocateRegistry.createRegistry(port);

                rmiRegistryLocation = rmiRegistryLocation + ":" + port;
                System.out.println("\n Active Registry Created RMI Registry At : " +
rmiRegistryLocation);

                IDomainSecurityManager dsmanager =
                        (IDomainSecurityManager) Naming.lookup(dsmLocation);
                AuthenticatedPacket authpacket = null;

                System.out.println("Active Registry Contacting DSM for Authentication.");

                authpacket =
                        dsmanager.authenticationService(
                                userType,
                                userName,
                                password,
                                activeRegistryLocation,
                                domain);

                System.out.println("Active Registry Authenticated by DSM.");

                MulticastReceiver mcastReceiver =
                        new MulticastReceiver(mcastPort, authpacket,
activeRegistryLocation);
                Thread receiverThread = new Thread(mcastReceiver);
                receiverThread.start();


} catch (Exception e) {
        System.out.println(e.getMessage());
}

} //end of constructor

/**
 * Returns components to HH.
 */
public Hashtable getComponentData() throws RemoteException
{
        Hashtable objectTable = new Hashtable();

        System.out.println("Active Registry contacted by Headhunter to Retrieve Component
Data");

        try
            {
                        String[] objURL = list();

                        for(int i=0;i<objURL.length;i++)
                    {
                                System.out.println("Active Registry gathering component
information from : " + objURL[i]);

                                //Registry looking up object registered with it.
                                System.setSecurityManager(new RMISecurityManager());
                                Object  obj = Naming.lookup(objURL[i]);

                                //Obtain the location(URL) of the UMMSpecification for
this object by
                                //introspecting its ummSpecification property.

                                String ummSpecURL = (String)
UniFrameIntrospector.getProperty(obj,"ummSpecURL");


                                //Call the XMLParser by passing this URL. The XML Parser
will parse the XML specification
                                //and construct a ComponentBean from the specification
which it returns.

                                UniFrameSpecificationParser xmlDomParser = new
UniFrameSpecificationParser(ummSpecURL);
                                ComponentBean componentBean =
xmlDomParser.getComponentBean();

                                //Add the component to the Hashtable
                                objectTable.put(objURL[i],componentBean);
                        }//end for

                }
                catch(Exception e){
                        System.out.println(e.getMessage());
                }

        //Once the Hashtable is filled return the hashtable
        return objectTable;
 }

}//end of Active Registry
```

**umm.services.DomainSecurityManager**

```
package umm.services;

import umm.entity.beans.*;
import umm.helper.dependent.*;
import umm.helper.dataaccess.*;

import java.net.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.security.Principal;
import java.security.acl.*;

import sun.security.acl.*;
import javax.crypto.*;

/**
 * This class implements IDomainSecurityManager interface.
 * The DSM performs the functions of generating secret-keys
 * for the domains using the CreateKey helper class.
 * It authenticates the principals against the DSM_Repository
 * with the help of the DSMRepositoryHelper and distributes the
 * secret-keys and multicast addresses to Headhunters and
 * Active Registries. The communication between the parties
 * is based on RMI-JRMP.
 * Creation date: (9/14/2001 12:57:07 PM)
 * @author: Nanditha Nayani Siram
 */

public class DomainSecurityManager extends UnicastRemoteObject implements
IDomainSecurityManager
{

        private static final String securityOwnerString = "DomainSecurityManager";
        private static final String aclIdentifier = "DomainSecurityACL";

        private static final Principal securityOwner = new
PrincipalImpl(securityOwnerString);
        private static final Acl domainACL = new AclImpl( securityOwner,aclIdentifier);

        private static Hashtable userdomainMapping = null;
        private static Hashtable domainList = null;
        private static Hashtable HHAddressAllocTable = new Hashtable();
        private static Hashtable registeredHHTable = new Hashtable();
        private static Hashtable keyTable = new Hashtable();
        private static Random rand = new Random();

/**
 * Construtor
 */

public DomainSecurityManager() throws RemoteException {
        super();
        try {
                createACLEntries();
                generateSecretKeys();
        } catch (DomainSecurityManagerException e) {
                System.out.println(e);
        }
}

public static void main(String[] args) {

        String dsmLocation = "//magellan.cs.iupui.edu:8500/DomainSecurityManager";
        try {
                System.setSecurityManager(new RMISecurityManager());
                Naming.rebind(dsmLocation, new DomainSecurityManager());
                System.out.println("DomainSecurityManager " + dsmLocation + " is ready.");

        } catch (Exception e) {
                System.out.println("DomainSecurityManager failed: " + e);
        }

}

/**
 * Return List of Registered HHs.
 */
 public java.util.ArrayList getHHListForDomain(String domainName)
        throws RemoteException {

    System.out.println("DSM Contacted by QM for HH List for : " + domainName + "
Domain");

        ArrayList hhList = new ArrayList();
        Enumeration e = registeredHHTable.keys();

        while (e.hasMoreElements()) {
                String key = (String) e.nextElement();
         String value = (String) registeredHHTable.get(key);
                if((value).equalsIgnoreCase(domainName))
         {
                        hhList.add(key);
         }//end if
        }//end while
        return hhList;
}

/**
 * authenticate principal against DSM_Repository
 */
private static Principal retrieveUser(
        String userType,
        String userName,
        String password)
        throws DomainSecurityManagerException {

        // load user from database
        boolean userExists =
                DSMRepositoryHelper.authenticateUser(userType, userName, password);

        // if not found, throw exception
        if (!userExists) {
                throw new DomainSecurityManagerException(
                        "User " + userName + " failed authentication.",
                        null);
        }

        System.out.println("DSM authenticated " + userName);

        // Create the Principal object
        Principal user = new PrincipalImpl(userName);

        return user;
}

/**
```

```
 * Get key for given domain.
 */
private static SecretKey getKey(String domainName)
        throws DomainSecurityManagerException {
        return (SecretKey) keyTable.get(domainName);
}

/**
 * Get multicast address for user.
 */
private static String getDomainAddressForUser(
        String userType,
        String userName,
        String password,
        String location,
        String domainName)
        throws DomainSecurityManagerException {
        String domainAddress = null;

        // Create the Principal object
        Principal user = retrieveUser(userType, userName, password);
        Permission permission = new PermissionImpl(domainName);
        if (isUserAuthenticated(user, permission)) {
                domainAddress = pickDomainAddress(userType, domainName, location);
        } else {
                throw new DomainSecurityManagerException(
                        "User " + userName + " is not authorized for the domain " +
domainName,
                        null);
        }

    System.out.println("DSM authorized " + userName +
                                " and allocated multicast address " + domainAddress);
        return domainAddress;
}

/**
 * Pick multicast address for HH/AR.
 */
private static String pickDomainAddress(
        String userType,
        String domainName,
        String location)
        throws DomainSecurityManagerException {
        String domainAddress = null;

        if ((domainList != null) && (domainList.containsValue(domainName))) {

                //If a registry requests an address check if there are headhunters in this
                //domain. Randomly pick an address in this domain to which headhunters
                //already multicast.
                if ((userType.equals("Registry"))
                        && (HHAddressAllocTable.containsValue(domainName))) {
                        Enumeration e = HHAddressAllocTable.keys();
                        Vector thisVector = new Vector();
                        while (e.hasMoreElements()) {
                                String key = (String) e.nextElement();
                                if (((String)
HHAddressAllocTable.get(key)).equals(domainName))
                                        thisVector.add(key);
                        }

                        int size = thisVector.size();
                        if (size > 0) {
                                int num = rand.nextInt(size);
                                domainAddress = (String) thisVector.get(num);
                        }
                }
                //If a headhunter requests an address randomly pick an address in this
```

```
domain to
               //allocate to headhunter.
               else {

                       Enumeration e = domainList.keys();
                       Vector thisVector = new Vector();
                       while (e.hasMoreElements()) {
                               String key = (String) e.nextElement();
                               if (((String) domainList.get(key)).equals(domainName))
                                       thisVector.add(key);
                       }

                       int size = thisVector.size();
                       if (size > 0) {
                               int num = rand.nextInt(size);
                               domainAddress = (String) thisVector.get(num);
                       }
                       //Add the allocated address to the headhunterList if not already
added.
                       if ((domainAddress != null)
                               && (userType.equals("HeadHunter"))
                               && !(HHAddressAllocTable.containsKey(domainAddress))) {
                               HHAddressAllocTable.put(domainAddress, domainName);
                               registeredHHTable.put(location, domainName);
                System.out.println("Registered " + (String)
registeredHHTable.get(location) +
                                 " Headhunter located at " + location);
                       }

               }
       } else {
               throw new DomainSecurityManagerException(
                       "No Such Domain Exists:  " + domainName,
                       null);
       }

       return domainAddress;
}

/**
 * Verify user authentication
 */
private static final boolean isUserAuthenticated(
       Principal user,
       Permission domain) {
       // Just use the java.security.acl.ACL class to handle this
       return domainACL.checkPermission(user, domain);
}

/**
 * add ACL Entries.
 */
private static void addAclEntry(String userName, String domain)
       throws DomainSecurityManagerException {

       // Create the Principal object
       Principal user = new PrincipalImpl(userName);

       // create a new Acl entry for this user
       AclEntry newAclEntry = new AclEntryImpl(user);

       // initialize some temporary variables
       Permission permission = new PermissionImpl(domain);

       // add the permission to the aclEntry
       newAclEntry.addPermission(permission);

       try {
               // add the aclEntry to the ACL for the securityOwner
```

```
                domainACL.addEntry(securityOwner, newAclEntry);

        } catch (NotOwnerException noE) {
                throw new DomainSecurityManagerException("In addAclEntry", noE);
        }
}

/**
 * Remote method called by HH/AR.
 */
public AuthenticatedPacket authenticationService(
        String userType,
        String userName,
        String password,
        String location,
        String domain)
        throws RemoteException {
        AuthenticatedPacket authPacket = new AuthenticatedPacket(null, null);
        try {
                authPacket.setMCAddress(
                        getDomainAddressForUser(
                                userType,
                                userName,
                                password,
                                location,
                                domain));
                authPacket.setKey(getKey(domain));

        } catch (Exception e) {
                System.out.println(e);
        }
        return authPacket;
}

/**
 * Create ACL Entries.
 */
private static void createACLEntries() throws DomainSecurityManagerException {

    System.out.println("DSM Creating ACL Entries. ");

        try {
                // loads all the domains addresses associated with
                // various domains into a hashtable
                if (userdomainMapping == null) {
                        // initialize the jdbc helper class
                        DSMRepositoryHelper.initialize();
                        userdomainMapping = DSMRepositoryHelper.loadUserDomainMapping();

                        if (userdomainMapping != null) {
                                Enumeration e = userdomainMapping.keys();
                                while (e.hasMoreElements()) {
                                        String key = (String) e.nextElement();
                                        String domainName = (String)
userdomainMapping.get(key);
                                        addAclEntry(key, domainName);
                                        System.out.println(
                                                "Added ACLEntry for User = " + key + "
Domain = " + domainName);

                                } //while
                        } //if(userdomainMapping != null)
                } //if (userdomainMapping==null)

                if (domainList == null) {
                        domainList = DSMRepositoryHelper.loadDomainList();
                        System.out.println("\n Loaded DomainList");
                }
        } catch (Exception e) {
                throw new DomainSecurityManagerException("Error in init method", e);
        }
}

/**
 * Generates secret Keys.
 */
private static void generateSecretKeys() throws DomainSecurityManagerException {

    System.out.println("DSM Generating Secret-Keys For Domains. ");
        try {
                ArrayList listOfDomains = DSMRepositoryHelper.getListOfDomains();

                for (int i = 0; i < listOfDomains.size(); i++) {
                        String domainName = (String) listOfDomains.get(i);
                        SecretKey key = CreateKey.getKey();
                        keyTable.put(domainName, key);
                        System.out.println("DSM Created Secret-Key for Domain : " +
domainName);
                } //end for
        } catch (Exception e) {
                throw new DomainSecurityManagerException(
                        "DSM failed to generate keys." + e,
                        null);
        }
}

}//end of DomainSecurityManager
```

**umm.services.DomainSecurityManager**

```
package umm.services;

public class DomainSecurityManagerException extends Exception
{

        private String message = null;
        private Exception exception = null;

        public DomainSecurityManagerException(String smessage, Exception ex)
        {
                // store the passed in values as class variables
                message = smessage;
                exception = ex;
        }

        public String getMessage()
        {
                // return the message to the user
                return message;
        }

        public void printStackTrace()
        {
                // output the message & print the StackTrace
                System.out.println( message);
                exception.printStackTrace();
        }
}
```

**umm.services.Headhunter**

```
package umm.services;

import umm.entity.beans.*;
import umm.helper.dependent.*;
import umm.services.*;
import umm.helper.dataaccess.*;
import java.net.*;
import java.util.*;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import java.security.*;
import java.io.*;

/**
 * This class implements the IHeadhunter  interface.
 * The Headhunter class interacts with the QueryManager
 * and DomainSecurityManager using RMI-JRMP. The Headhunter
 * uses the MulticastSender to multicast encrypted messages
 * to ActiveRegistry services. The Headhunter also communicates
 * with the ActiveRegistry services through unicast RMI-JRMP for
 * the purpose of obtaining the component information as a Hashtable
 * of ComponentBean objects. The HeadHunter persists this component
 * information to the Meta_Repository and uses the MetaRepositoryHelper
 * for the purpose of performing searches against the repository.
 * The SQL query for the searches is obtained from the QueryBean which
 * is propagated to the Headhunter by the QueryManager.
 * Creation date: (10/15/2001 11:50:10 AM)
 * @author: Nanditha Nayani
 */

public class Headhunter extends UnicastRemoteObject implements IHeadhunter
{

private Hashtable registryTable = new Hashtable();
private java.lang.String userType = "HeadHunter";

/**
* Remote method invoked by QM.
*/
public Hashtable performSearch(QueryBean querybean) throws RemoteException {

    System.out.println("Processing Query Request From Query Manager");

       MetaRepositoryHelper srchEngine = new MetaRepositoryHelper(querybean);
       try {
               return srchEngine.getSearchResultTable();
       } catch (Exception e) {
               System.out.println(e);
               return null;
       }
}

/**
* Create the MR.
*/
private static void createMetaRepository() throws Exception {


       String componentCreateString =
       "CREATE TABLE COMPONENT(" +
       "ID  VARCHAR2(150) PRIMARY KEY," +
       "NAME VARCHAR2(100)," +
       "DESCRIPTION VARCHAR2(1000)," +
       "FUNCTION VARCHAR2(500)," +
       "ALGORITHM VARCHAR2(200)," +
       "COMPLEXITY VARCHAR2(30)," +
       "DOMAIN VARCHAR2(30)," +
       "TECHNOLOGY VARCHAR2(30)," +
       "COLLABORATORS VARCHAR2(200)," +
       "END2ENDDELAY NUMBER(9)," +
       "AVAILABILITY NUMBER(9)," +
        "MOBILITY VARCHAR2(5))";

        String functionCreateString =

        "CREATE TABLE FUNCTION(" +
        "ID  VARCHAR2(150) NOT NULL," +
        "FUNCTION_NAME VARCHAR2(250) NOT NULL," +
        "SYNTACTIC_CONTRACT VARCHAR2(250) NOT NULL," +
        "CONSTRAINT pk1 PRIMARY KEY(ID,FUNCTION_NAME,SYNTACTIC_CONTRACT)," +
        "CONSTRAINT fk1 FOREIGN KEY(ID)" +
        "REFERENCES COMPONENT(ID) ON DELETE CASCADE)";

        SQLHelper sqlEngine = new SQLHelper();
        sqlEngine.updateTable(componentCreateString);
        sqlEngine.updateTable(functionCreateString);
        sqlEngine.shutDown();

}

public static void main(String[] args) {

        long mcastTime = 5000;
        String headhunterLocation = "//magellan.cs.iupui.edu:8500/HeadHunter";
        String dsmLocation = "//magellan.cs.iupui.edu:8500/DomainSecurityManager";
        int mcastPort = 10000;
        String userName = "HeadHunt1";
        String password = "HeadHunt1";
        String domain = "Finance";

        if(args.length > 0)
           mcastTime = Long.parseLong(args[0]);

        try {
                System.setSecurityManager(new RMISecurityManager());
                Naming.rebind(
                        headhunterLocation,
                        new Headhunter(
                            mcastTime,
                                mcastPort,
                                userName,
                                password,
                                domain,
                                headhunterLocation,
                                dsmLocation));
                System.out.println("HeadHunter is ready.");
        } catch (Exception e) {
                System.out.println("HeadHunter failed: " + e);
        }

}

/**
* Constructor
*/
 public Headhunter(
    long mcastTime,
        int mcastPort,
        String userName,
        String password,
        String domain,
        String headhunterLocation,
        String dsmLocation)
        throws RemoteException {

        System.out.println("\n HeadHunter activated at " + headhunterLocation);

        try {
                Headhunter.createMetaRepository();
                System.out.println("MetaRepository Created.");
        } catch (Exception e) {
                //Ignore if repository already created
                System.out.println("MetaRepository Created.");
        }



        try {

                System.out.println("Headhunter Contacting DSM for Authentication.");

                IDomainSecurityManager dsmanager =
                        (IDomainSecurityManager) Naming.lookup(dsmLocation);
                AuthenticatedPacket authpacket = null;
                authpacket =
                        dsmanager.authenticationService(
                                userType,
                                userName,
                                password,
                                headhunterLocation,
                                domain);

                System.out.println("Headhunter Authenticated by DSM");

                MulticastSender mcastSender =
                        new MulticastSender(mcastTime, mcastPort, authpacket,
headhunterLocation);
                Thread senderThread = new Thread(mcastSender);
                senderThread.start();

        } catch (Exception e) {
                System.out.println(e.getMessage());
        }

} //end of constructor

/**
 * Persist component information obtained from AR
 * into MR.
 */
private void populateMetaRepository(String regLoc) {

        // retrieve component info
        try {

                SQLHelper sqlEngine = new SQLHelper();

                System.out.println("Headhunter contacting Registry at " + regLoc + " for
registered services data.");
                System.setSecurityManager(new RMISecurityManager());
                IActiveRegistry Reg = (IActiveRegistry) Naming.lookup(regLoc.trim());

                Hashtable CompData = (Hashtable) Reg.getComponentData();
                System.out.println("Headhunter obtained registered services data from : "
+ regLoc);

                Enumeration e = CompData.elements();
                while (e.hasMoreElements()) {
                        ComponentBean componentBean = (ComponentBean) e.nextElement();
                        try {
                                componentBean.persistBean(sqlEngine);
                        } catch (Exception ex) {
                                //If the component already exists in MR then ignore.
                        }
                        System.out.println("Headhunter persisted component " +
componentBean.getId() + " into MR.");
                }//end while

                sqlEngine.shutDown();
        } catch (Exception ex) {
                System.out.println("HH exception " + ex);
        }
}

/**
* Remote method invoked by AR to pass its location information
*/
public void receiveUnicastCommunication(String regLoc) throws RemoteException {

        System.out.println("Headhunter received unicast communication from Registry at : "
+ regLoc);
        registryTable.put(regLoc, (new java.util.Date()));
        populateMetaRepository(regLoc);

}

}//end of HeadHunter
```

**umm.services.IActiveRegistry**

```
package umm.services;

import java.rmi.*;
import java.util.*;

/**
 * This is the interface for the ActiveRegistry service.
 * This interface publishes the following method:
 * getComponentData(): This method is invoked by the Headhunter
 * to retrieve component information from the ActiveRegistry.
 * Creation date: (9/14/2001 12:57:07 PM)
 * @author: Nanditha Nayani Siram
 */
public interface IActiveRegistry extends Remote
{
        public Hashtable getComponentData() throws RemoteException;
}
```

**umm.services.IDomainSecurityManager**

```
package umm.services;

import umm.entity.beans.*;

import java.util.*;
import java.rmi.*;

/**
 * This is the interface for the DomainSecurityManager service.
 * This interface publishes two methods:
 * getHHListForDomain: This method is invoked by the QueryManager.
 * The purpose of this method is to return a list of registered
 * Headhunters for a particular domain to the QueryManager.
 * authenticationService: This method is invoked by the Headhunter
 * and ActiveRegistry components to authenticate themselves with
 * the DSM.
 * Creation date: (3/16/02 9:07:49 PM)
 * @author:
 */
public interface IDomainSecurityManager extends java.rmi.Remote{

        public ArrayList getHHListForDomain(String domainName) throws RemoteException;
        public AuthenticatedPacket authenticationService(String userType, String userName,
String password, String contactLocation, String domain) throws RemoteException;
}
```

**umm.services.IHeadhunter**

```
package umm.services;
```

```
import umm.entity.beans.*;

import java.rmi.*;
import java.util.*;
import java.security.*;
import java.io.*;

/**
 * This is the interface for the Headhunter service.
 * This interface publishes the following methods:
 * performSearch: This method is invoked by the QueryManager
 * to propagate a search query.
 * receiveUnicastCommunication: This method is invoked by the
 * ActiveRegistry to inform the Headhunter of its location.
 * Creation date: (9/14/2001 12:57:07 PM)
 * @author: Nanditha Nayani Siram
 */
public interface IHeadhunter extends Remote {

        public Hashtable performSearch(QueryBean querybean) throws RemoteException;
        public void receiveUnicastCommunication(String regLoc) throws RemoteException;

}
```

**umm.services.QueryManager**

```
package umm.services;

import umm.entity.beans.*;
import umm.services.*;

import java.net.*;
import java.util.*;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;


/**
 * This class implements the IQueryManager interface.
 * The QueryManager class propagates the search query
 * as a QueryBean object to the list of Headhunter
 * components obtained from DomainSecurityManager and
 * returns search results to the RequestProcessor.
 * The interaction between these components is based
 * on RMI-JRMP.
 * Creation date: (10/15/2001 11:50:10 AM)
 * @author: Nanditha Nayani Siram
 */

public class QueryManager extends UnicastRemoteObject implements IQueryManager
{

   private IDomainSecurityManager dsm = null;

/**
 * Constructor
 */
public QueryManager(String dsmLocation) throws RemoteException {

        try {
                System.setSecurityManager(new RMISecurityManager());
                dsm = (IDomainSecurityManager) Naming.lookup(dsmLocation);

        } catch (Exception e) {
                System.out.println(e.getMessage());
        }

} //end of constructor
/**
 * Remote Method invoked by RequestProcessor to
 * retrieve search results.
 */
public Hashtable getSearchResultTable(QueryBean querybean)
        throws RemoteException {

        System.out.println("QM Requested to propagate Search Query.");

        Hashtable resultTable = new Hashtable();

        System.setSecurityManager(new RMISecurityManager());
        ArrayList hhList = dsm.getHHListForDomain(querybean.getDomain());
    System.out.println("QM obtained regsitered headhunter list from DSM for Domain " +
                              querybean.getDomain());

        for (int i = 0; i < hhList.size(); i++)
    {
                try {

                        String hhLocation = (String) hhList.get(i);

                        IHeadhunter hh = (IHeadhunter) Naming.lookup(hhLocation.trim());

                        System.out.println("QM Propagating Query To HH At " + hhLocation);
                        Hashtable thisResultTable = hh.performSearch(querybean);
            System.out.println("QM Obtained Results from HH At " + hhLocation);

                        Enumeration e = thisResultTable.keys();
                        while (e.hasMoreElements()) {
                                String id = (String) e.nextElement();
                                ComponentBean cbean = (ComponentBean)
thisResultTable.get(id);
                                resultTable.put(id, cbean);

                        } //end while

                } catch (Exception e) {
                        //Ignore exception and move on to next headhunter.
                }
        } //end for

        return resultTable;

}

public static void main(String[] args)
{

         String dsmLocation = "//magellan.cs.iupui.edu:8500/DomainSecurityManager";
         String qmLocation = "//magellan.cs.iupui.edu:8500/QueryManager";
         try
        {
                        System.setSecurityManager(new RMISecurityManager());
                        Naming.rebind (qmLocation, new QueryManager(dsmLocation));
                        System.out.println ("QueryManager is ready.");
        }
         catch (Exception e)
        {
                        System.out.println ("QueryManager failed: " + e);
        }

}


}//end of QueryManager
```

**umm.tests.AccountServer**

```
package umm.tests;

import java.rmi.server.*;
import java.rmi.*;

/**
 * This is the test service component. The class
 * specifies the ummSpecURL property and has
 * a getter which return the URL.
 * Creation date: (9/14/2001 12:57:07 PM)
 * @author: Nanditha Nayani Siram
 */

public class AccountServer extends UnicastRemoteObject implements IAccountServer{

         private String ummSpecURL = "";

public AccountServer(String newUMMSpecURL) throws RemoteException{
        ummSpecURL = newUMMSpecURL;
}

public String getUmmSpecURL() throws RemoteException {
        return ummSpecURL;
}

public void javaDeposit(float arg1)throws RemoteException { }
public void javaWithdraw(float arg1) throws RemoteException { }
public float javaBalance() throws RemoteException{ return 0;}

public static void main(String[] args) {

        String bindingName = "//magellan.cs.iupui.edu:9000/AccountServer" + args[0];
        String ummSpecUrl = "/home/nnayani/ummspecs/ummspec" + args[1] + ".xml";

        try {
               System.setSecurityManager(new RMISecurityManager());
               Naming.rebind(bindingName, new AccountServer(ummSpecUrl));
               System.out.println("Account Server : " + bindingName + " is ready.");
        } catch (Exception e) {
               System.out.println("Account Server failed: " + e);
        }
}
}
```

**umm.tests.URDSClient**

```
package umm.tests;

import umm.entity.beans.*;
import umm.helper.dependent.*;

import java.util.*;

/**
 * This is a Test Application Client.
 * The client issues several different types and
 * queries via the Request Processor.
 * Creation date: (3/17/02 5:40:28 AM)
 * @author: Nanditha Nayani Siram
 */
public class URDSClient {

/**
 * urdsClientDriver constructor comment.
 */
public URDSClient() {
        super();
}
/**
 * Insert the method's description here.
 * Creation date: (3/17/02 5:55:09 AM)
 * @param args java.lang.String[]
 */
public static void main(String[] args) {

        URDSClient urdsClient = new URDSClient();

        //Test Queries
        QueryBean queryBean1 = urdsClient.Query1();
        QueryBean queryBean2 = urdsClient.Query2();
        QueryBean queryBean3 = urdsClient.Query3();
        QueryBean queryBean4 = urdsClient.Query4();
        QueryBean queryBean5 = urdsClient.Query5();
        QueryBean queryBean6 = urdsClient.Query6();
        QueryBean queryBean7 = urdsClient.Query7();
        QueryBean queryBean8 = urdsClient.Query8();
        QueryBean queryBean9 = urdsClient.Query9();
        QueryBean queryBean10 = urdsClient.Query10();

        ArrayList queryList = new ArrayList();
        queryList.add(queryBean1);
    queryList.add(queryBean2);
        queryList.add(queryBean3);
        queryList.add(queryBean4);
        queryList.add(queryBean5);
        queryList.add(queryBean6);
        queryList.add(queryBean7);
        queryList.add(queryBean8);
        queryList.add(queryBean9);
        queryList.add(queryBean10);

        long counter=0;
        long timer=0;

    //Iterate 10 times through 10 queries (10 * 10 =100 queries)
    for(int j=0;j<10;j++)
    {
                for(int i=0;i<queryList.size();i++)
                {
                        //Increment counter for each new query request.
                        counter++;
                        try {

                 //Get each new query
                                QueryBean queryBean = (QueryBean) queryList.get(i);
                                System.out.println(queryBean.getQuery() + "\n");

                 //Instantiate RequestProcessor and pass it the query.
                                RequestProcessor reqProcessor = new RequestProcessor();
                 reqProcessor.setQueryBean(queryBean);

                 //starting the timer before query progagation.
                                long startTime = System.currentTimeMillis();
                                //Actual call to propagate client query.
                                Hashtable resultTable = reqProcessor.getResultTable();
                                //Ending the timer.
                                long endTime = System.currentTimeMillis();

                                //Computing the net time for the query servicing so far.
                                timer = timer + (endTime - startTime);

                                urdsClient.printResults(resultTable);

                        } catch (Exception e) {
                                System.out.println(e);
                        }
                }//end for
    }//end iteration

    long avgServiceTime = timer/counter;
        System.out.println("Number of iterations of Client Queries Services = " +
counter);
    System.out.println("Avergae time taken for servicing a Client Query = " +
avgServiceTime);

}

/**
 * Print Results.
 */
public void printResults(Hashtable resultTable) {

     if(resultTable != null)
     {
                Enumeration e = resultTable.keys();
                while(e.hasMoreElements())
                {
                        String key = (String) e.nextElement();
                        ComponentBean cbean = (ComponentBean) resultTable.get(key);
                        System.out.println(cbean.getId());
                }
                System.out.println("---------------");

     }
}

public QueryBean Query1() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
        return queryBean;
}

public QueryBean Query2() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
         queryBean.setAvailabilityFlag(true);
         queryBean.setAvailabilityConstraint("<");
         queryBean.setAvailabilityValue(100);
        return queryBean;
}

public QueryBean Query3() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setComponentName("Account");
        return queryBean;
}

public QueryBean Query4() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setComponentDescription("Account Management");
        return queryBean;
}

public QueryBean Query5() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setComplexity("O(1)");
        return queryBean;
}

public QueryBean Query6() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
         queryBean.setEnd2endDelayFlag(true);
         queryBean.setEnd2endDelayConstraint("<");
         queryBean.setEnd2endDelayValue(50);
        return queryBean;
}

public QueryBean Query7() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setFunctionNames("deposit withdraw");
        return queryBean;
}

public QueryBean Query8() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setMobility("yes");
        return queryBean;
}

public QueryBean Query9() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setTechnology("Java-RMI");
        return queryBean;
}

public QueryBean Query10() {
        QueryBean queryBean = new QueryBean();
        queryBean.setDomain("Finance");
    queryBean.setAlgorithms("add sub");
        return queryBean;
}

}//end of URDSClient
```

**UniFrameQuery.jsp**

```
<html>
<html>
<head>
<meta http-equiv="PRAGMA" content="NO-CACHE">
</head>
<%@ page import="java.util.*" %>
<%@ page isErrorPage="false" errorPage="error.jsp"%>
<body bgcolor = #F8F7D9>
<form name="queryFilterForm" method="post" action="ProcessUniFrameQuery.jsp">

  <table width="774" border="0" cellspacing="1" cellpadding="0">
    <tr>
    <td bgcolor="#CCCCCC" valign="top" height="19" colspan="5" >
        <b> Search By Component Details </b>
        </td>
      <td width="5"></td>
     </tr>

     <tr>
      <td height="0" width="104"></td>
      <td width="137"></td>
      <td width="60"></td>
      <td width="277"></td>
      <td width="184"></td>
      <td width="5"></td>
    </tr>
    <tr valign="top">

      <td height="21" colspan="2"><b>Domain</b></td>
      <td valign="top" colspan="4" height="21">
        <select name="domain">
                <option value="Finance" selected>Finance</option>
                <option value="Manufacturing">Manufacturing</option>
              </select>
    </td>
    </tr>
    <tr>

      <td height="37" valign="top" colspan="2"><b>Component Name
        (Enter Keywords)</b></td>

      <td colspan="3" valign="top" height="37">
        <input type="text" name="componentName" size="80" value="">
            </td>

      <td width="5" height="37"></td>
          </tr>
          <tr>

      <td height="40" valign="top" colspan="2"><b>Component Description <br>
        (Enter Keywords)</b> </td>

      <td colspan="3" valign="top" height="40">
        <input type="text" name="componentDescription" size="80" value="">
            </td>

      <td width="5" height="40"></td>
          </tr>
          <tr>

      <td height="37" valign="top" colspan="2">
        <p><b>Function Names<br>
          </b><b>(Enter Keywords) </b></p>
        </td>

      <td colspan="3" valign="top" height="37">
        <input type="text" name="functionNames" size="80" value="">
            </td>

      <td width="5" height="37"></td>
          </tr>
                 <tr>

      <td bgcolor="#CCCCCC" valign="top" height="19" colspan="5" ><b>Search By Functional
Attributes
        </b></td>
      <td width="5"></td>
          </tr>
          <tr>

      <td height="38" valign="top" colspan="2"><b>Desired Algorithms <br>
        (Enter Keywords)</b> </td>

      <td colspan="3" valign="top" height="38">
        <input type="text" name="algorithms" size="80" value="">
            </td>

      <td width="5" height="38"></td>
          </tr>
          <tr>

      <td height="40" valign="top" colspan="2"><b>Desired Complexity <br>
        (Enter Keywords) </b></td>

      <td colspan="3" valign="top" height="40">
        <input type="text" name="complexity" size="80" value="">
            </td>
```

```
<td width="5" height="40"></td>
    </tr>
    <tr valign="top">

<td height="30" colspan="2"><b>Technology</b></td>
      <td valign="top" colspan="4">
        <select name="technology">
          <option value="" selected>None</option>
          <option value="Java-RMI">Java-RMI</option>
          <option value="CORBA">CORBA</option>
          <option value="Voyager">Voyager</option>
        </select>
      </td>
    </tr>
    <tr>

<td bgcolor="#CCCCCC" valign="top" height="19" colspan="5" ><b>Search By
  Auxillary Atributes</b></td>

<td width="5"></td>
    </tr>
           <tr valign="top">

<td height="30" colspan="2"><b>Mobility</b></td>
      <td valign="top" colspan="4">
        <select name="mobility">
          <option value="No" selected>No</option>
          <option value="Yes">Yes</option>
        </select>
      </td>
    </tr>
    <tr valign="top">

<td bgcolor="#CCCCCC" colspan="5" valign="top" height="19" ><b>Search By
  QOS Metrics</b></td>

<td width="5"></td>
    </tr>
    <tr>

<td height="15" valign="top" bgcolor="#CCCCCC" width="104"><b>Select</b></td>

<td colspan="2" valign="top" bgcolor="#CCCCCC"><b>QOS Parameter</b></td>

<td valign="top" bgcolor="#CCCCCC" width="277"><b>Constraints</b></td>

<td valign="top" bgcolor="#CCCCCC" width="184"><b>Preferences</b></td>

<td width="5"></td>
    </tr>
    <tr>

<td height="41" valign="top" width="104">
  <input type="checkbox" name="qosMetric" value="end2endDelay">
      </td>

<td colspan="2" valign="top"> <b>End To End Delay</b> </td>

<td valign="top" width="277">
  <select name="end2endDelayConstraint">
          <option value="None" selected>None</option>
          <option value="=">=</option>
          <option value="<">&lt;</option>
          <option value=">">&gt;</option>
          <option value="<=">&lt;=</option>
          <option value=">=">&gt;=</option>
        </select>
        <input type="text" name="end2endDelayValue" size="20" value="">
      </td>

<td valign="top" width="184">
  <select name="end2endDelayPreference">
          <option value="None" selected>None</option>
          <option value="Descending">Max</option>
          <option value="Ascending">Min</option>
          <option value="With">With</option>
          <option value="Random">Random</option>
          <option value="First">First</option>
        </select>
      </td>

<td width="5"></td>
    </tr>
            <tr>

<td height="41" valign="top" width="104">
  <input type="checkbox" name="qosMetric" value="availibility">
      </td>

<td colspan="2" valign="top"> <b>Availability</b></td>

<td valign="top" width="277">
  <select name="availabilityConstraint">
          <option value="None" selected>None</option>
          <option value="=">=</option>
          <option value="<">&lt;</option>
          <option value=">">&gt;</option>
          <option value="<=">&lt;=</option>
          <option value=">=">&gt;=</option>
        </select>
        <input type="text" name="availabilityValue" size="20" value="">
      </td>

<td valign="top" width="184">
  <select name="availabilityPreference">
          <option value="None" selected>None</option>
          <option value="Descending">Max</option>
          <option value="Ascending">Min</option>
          <option value="With">With</option>
          <option value="Random">Random</option>
          <option value="First">First</option>
        </select>
      </td>

<td width="5"></td>
    </tr>
    <tr valign="top">

<td bgcolor="#CCCCCC" colspan="5" valign="top" height="19" > <b>Specify
  Policies</b></td>

<td width="5"></td>
    </tr>
    <tr>

<td height="18" colspan="6" valign="top" bgcolor="#CCCCCC"><b>Search Scoping
  Policies</b></td>
    </tr>
           <tr>

<td height="35" valign="top" colspan="2"><b>Max Upper Bound Of Offers To
  Return</b> </td>

<td colspan="3" valign="top" height="35">
  <input type="text" name="numOffers" size="80" value="">
      </td>
      <td width="5" height="35"></td>
          </tr>
                 <tr>

      <td height="22" colspan="6" valign="top" bgcolor="#CCCCCC"><b>Function Scoping
        Policies</b></td>
          </tr>
                 <tr>

      <td height="36" valign="top" colspan="2"><b>Min Number Of QOS Metrics To
        Match</b> </td>

      <td colspan="3" valign="top" height="36">
        <input type="text" name="numMetrics" size="80" value="">
            </td>

      <td width="5" height="36"></td>
          </tr>
        </table>
        <p align="center">
          <input type="submit" name="SubmitForm" value="Perform Search">
          <input type="reset" name="Submit2" value="Reset Form">
        </p>
      </form>
 </body>
 </html>
```

**ProcessUniframeQuery.jsp**

```
<%@ page isErrorPage="false" errorPage="error.jsp"%>

<jsp:useBean id="reqProcessor" class="umm.helper.dependent.RequestProcessor"
scope="session" />
<jsp:useBean id="queryBean" class="umm.entity.beans.QueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="*"/>

<%
  String[] qosMetrics = request.getParameterValues("qosMetric");
  if(qosMetrics != null)
  {
     for(int i = 0;i<qosMetrics.length;i++)
     {
      if(qosMetrics[i].equals("end2endDelay"))
           queryBean.setEnd2endDelayFlag(true);
         else
         if(qosMetrics[i].equals("availability"))
           queryBean.setAvailabilityFlag(true);
     }
  }
%>

<% reqProcessor.setQueryBean(queryBean); %>
<jsp:forward page="componentList.jsp" />
```

**ComponentList.jsp**

```
<html>
<head>
<meta http-equiv="PRAGMA" content="NO-CACHE">
</head>
<body bgcolor = #F8F7D9>
<%@ page import="java.util.*" %>
<%@ page import="umm.entity.beans.ComponentBean" %>
<%@ page import="umm.entity.beans.FunctionBean" %>
<%@ page import="umm.entity.beans.QueryBean" %>
<jsp:useBean id="reqProcessor" class="umm.helper.dependent.RequestProcessor"
scope="session" />
<%@ page isErrorPage="false" errorPage="error.jsp"%>
```

```
<table width="100%" border="0" cellspacing="1" cellpadding="0" height="83">
    <tr>

    <td bgcolor="#CCCCCC" colspan="4" valign="top" >
        <b>URDS Search Results </b>
       </td>
        </tr>
        <tr>

    <td valign="top" bgcolor="#CCCCCC"><b>Component-ID</b></td>

    <td valign="top" bgcolor="#CCCCCC"><b>Component Name</b></td>

    <td valign="top" bgcolor="#CCCCCC"><b>Component Details</b></td>
        </tr>
        <tr>

<%

   Hashtable resultTable = reqProcessor.getResultTable();
   if(resultTable != null)
   {
      Enumeration e = resultTable.keys();
          if(e.hasMoreElements())
          {
               while(e.hasMoreElements())
               {
                   String key = (String) e.nextElement();
                       ComponentBean componentBean = (ComponentBean) resultTable.get(key);

%>



    <td valign="top" height="32"><a
href="componentDetail.jsp?id=<%=componentBean.getId()%>"><%=componentBean.getId()%></a></t
d>

    <td valign="top"><a
href="componentDetail.jsp?id=<%=componentBean.getId()%>"><%=componentBean.getName()%></a><
/td>

    <td valign="top"><a href="componentDetail.jsp?id=<%=componentBean.getId()%>">
      Component Details</a></td>
        </tr>
<%
             }//end while
      }//end of if e.hasMoreElements
     else
     {
        throw new Exception("No Results Matching Search Criteria");
     }//if it does not have elements

  }else
  {
      throw new Exception("No Results Matching Search Criteria");
  }//if resultTable is null

%>
      </table>
          <br>

<a href = "UniFrameQuery.jsp">Search Home </a>
</body>
</html>
```

**ComponentDetail.jsp**

```
<html>
<head>
```

```
<meta http-equiv="PRAGMA" content="NO-CACHE">

</head>
<body bgcolor = #F8F7D9>
<%@ page import="java.util.*" %>
<%@ page import="umm.entity.beans.ComponentBean" %>
<%@ page import="umm.entity.beans.FunctionBean" %>
<%@ page import="umm.entity.beans.QueryBean" %>
<%@ page isErrorPage="false" errorPage="error.jsp"%>
<jsp:useBean id="reqProcessor" class="umm.helper.dependent.RequestProcessor"
scope="session" />

<%
   Hashtable resultTable = reqProcessor.getResultTable();
   if(resultTable != null)
   {
        String id = request.getParameter("id");
       ComponentBean componentBean = (ComponentBean) resultTable.get(id);
%>
<table width="100%" border="0" cellspacing="0" cellpadding="0" height="530">
  <tr>
    <td bgcolor="#CCCCCC" colspan="3" height="24" ><b>
      Component Details Summary </b></td>
  </tr>
   <tr valign="top">
    <td colspan="2" height="27"><b>Name</b></td>
    <td width="929" height="27"><%=componentBean.getName()%></td>
  </tr>
  <tr valign="top">
    <td width="202" valign="top" nowrap><b>Description</b></td>
    <td valign="top" colspan="2">
      <textarea name="projectDescription" rows="3" cols="40"
onFocus="JavaScript:this.blur()"><%=componentBean.getDescription()%></textarea>
    </td>
  </tr>
  <tr valign="top">
    <td colspan="3" height="17" valign="top" bgcolor="#CCCCCC" ><b>Computational
      Attributes</b></td>
  </tr>
  <tr valign="top">
    <td height="18" colspan="2" valign="top" ><b>ID</b></td>
    <td valign="top" width="929" ><%=componentBean.getId()%></td>
  </tr>
  <tr valign="top">
    <td height="20" colspan="3" valign="top" bgcolor="#CCCCCC" ><b>Cooperating
      Attributes</b></td>
  </tr>
  <tr valign="top">
    <td height="22" colspan="2" valign="top" ><b>Pre-Processing <br>
Collaborator(s)</b></td>
    <td valign="top" width="929" > <%=componentBean.getPreprocessingCollaborators()%>
    </td>
  </tr>
  <tr valign="top">
    <td height="20" colspan="3" valign="top" bgcolor="#CCCCCC" ><b>Auxillary
Attributes</b></td>
  </tr>
  <tr valign="top">
    <td height="22" colspan="2" valign="top" ><b>Mobility</b></td>
    <td valign="top" width="929" > <%=componentBean.getMobility()%> </td>
  </tr>
  <tr valign="top">
    <td height="18" colspan="3" valign="top" bgcolor="#CCCCCC" ><b>QOS Metrics</b></td>
  </tr>
  <tr valign="top">
    <td height="18" colspan="2" valign="top" ><b>End To End Delay</b></td>
    <td valign="top" width="929" > <%=componentBean.getEnd2endDelay()%> </td>
  </tr>
  <tr valign="top">
```

```
    <td height="22" colspan="2" valign="top" ><b>Availability</b></td>
    <td valign="top" width="929" > <%=componentBean.getAvailability()%> </td>
  </tr>
  <tr valign="top">
    <td height="20" colspan="3" valign="top" bgcolor="#CCCCCC" ><b>Functional
      Attributes</b></td>
  </tr>
  <tr valign="top">
    <td height="22" colspan="2" valign="top" nowrap ><b>Technology</b></td>
    <td valign="top" nowrap width="929" ><%=componentBean.getTechnology()%></td>
  </tr>
  <tr valign="top">
    <td height="20" colspan="2" valign="top" ><b>Algorithms</b></td>
    <td valign="top" width="929" ><%=componentBean.getAlgorithm()%></td>
  </tr>
  <tr valign="top">
    <td height="20" colspan="2" valign="top" ><b>Complexity</b></td>
    <td valign="top" width="929" ><%=componentBean.getComplexity()%></td>
  </tr>
  <tr valign="top">
    <td height="85" colspan="3" valign="top" >
      <table width="100%" border="0" cellpadding="0" cellspacing="0">
        <tr>
          <td valign="top" height="26" bgcolor="#CCCCCC"><b>Function</b></td>
          <td valign="top" bgcolor="#CCCCCC"><b>Syntactic Contract</b></td>
        </tr>
        <%
         Vector functionVector = componentBean.getFunctionBeanList();
         int size = 0;
         if(!functionVector.isEmpty())
         {
                size = functionVector.size();

        for(int j=0;j<size;j++) {
                FunctionBean functionBean =(FunctionBean) functionVector.elementAt(j);
        %>
        <tr>
          <td valign="top"> <%=functionBean.getFunctionName()%>
          </td>
          <td valign="top"> <%=functionBean.getSyntacticContract()%>
          </td>
        </tr>
        <%     } //end of for
          }//end of if
         %>
      </table>
    </td>
  </tr>
</table>
<%
  }//end if (results != null)
%>
<br>
<a href = "UniFrameQuery.jsp">Search Home </a>
<br>
<a href = "componentList.jsp">Back To Component List </a>
</body>
</html>
```

**Error.jsp**

```
<html>
<head>
<meta http-equiv="PRAGMA" content="NO-CACHE">
</head>

<body bgcolor = #F8F7D9 text = #098D3A>
<%@ page isErrorPage="true" %>
      <table width="100%" border="0" cellspacing="0" cellpadding="1" align="center">
```

```
        <tr>
          <td bgcolor="#CCCCCC" valign="top" height="19" colspan="5" > <b>Error
Page</b></td>
        </tr>
        <tr>
          <td valign="top" height="128"> <br>
             <b> <%= exception.getMessage() %> </b>
                        <br>
          </td>
        </tr>
       </table>

<a href = "UniFrameQuery.jsp">Search Home </a>
</body>
</html>
```

## LIST OF REFERENCES

[ACC97] “Access Control Abstractions”, May 1997.
http://java.sun.com/products/jdk/1.1/docs/guide/security/Acl.html

[APA] Apache HTTP Server Project, “Apache HTTP Server Version 1.3”
Http://httpd.apache.org/

[BAL95] Ballardie, T.,Crowcroft, J., “Multicast-specific Security Threats and Counter-measures”, In Proc. Symposium on Network and Distributed System Security, pages 2-16, San Diego,California, February 1995.

[BAL96] Ballardie, T., “Scalable Multicast Key Distribution”, RFC 1949, May 1996.

[BAR00] Barrett R. B., “Object-Oriented Natural Language Requirements Specification”, In Proceedings of ACSC 2000,the 23rd Australasian Computer Science Conference, January 31-February 4, 2000, Canberra, Australia, pages 24-30, January 2000.

[BEA] BEA, “BEA WebLogic Server 7.0”.
http://www.bea.com/products/weblogic/server/index.shtml

[BLU] Bluetooth Consortium, "The Bluetooth Consortium," http://www.bluetooth.com

[BLU97] Blundo, C., De Santis, A., Herzberg, A. , Kutten, S., Vaccaro, U.,Yung, M., “Perfectly-Secure Key Distribution for Dynamic Conferences. Information and Computation”, December 1997.

[BOX00] Box, D., et al., “Simple Object Access Protocol (SOAP) 1.1”, W3C, May 2000,http://www.w3.org/TR/SOAP.

[BRA00] Bray, T., Paoli, J., Sperberg-McQueen, C. M. “Extensible Markup Language (XML) 1.0 (Second Edition),” W3C, October 2000, http: //www.w3c.org/xml.

[BRA02] Brahmnath, G., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., “A Quality of Service Catalog for Software Components,” to appear in Proceedings of the 2002 Southeastern Software Engineering Conference, 2002.

[CAR98] Caronni, G., Waldvogel, M., Sun, D., Plattner, B., “Efficient Security for Large and Dynamic Groups”. Technical Report TIK Technical Report No. 41, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, February 1998.

[CHR01] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., “Web Services Description Language (WSDL) 1.1,” W3C, March 2001 http://www.w3.org/TR/wsdl.

[CHR99] Christensson, Bengt and Larsson, Olof, “Universal Plug and Play Connects Smart Devices,” WinHEC 99, 1999. http://www.axis.com/products/documentation/UPnP.doc

[COU01] Coulouris,G., Dollimore,J., Kindberg,T., “Distributed Systems Concepts And Design.”, Addison-Wesley Publishers Limited (Third Edition) 2001.

[CZA00] Czarnecki, K., Eisenecker, U.W, “Generative Programming Methods, Tools, and Applications”, Addison-Wesley, 2000.

[CZE99] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., Katz, R. H., “An Architecture for a Secure Service Discovery Service,” Proceedings of Mobicom '99, 1999. http://ninja.cs.berkeley.edu/dist/papers/sds-mobicom.pdf

[DES77] Data Encryption Standard, National Bureau of Standards Federal Information Processing Standard (FIPS) Publication 46, “Data Encryption Standard”, U.S. Department of Commerce, January 1977.

[DON99] Dondeti, R.,L., Mukherjee, S., Samal., A., “A Dual Encryption Protocol for Scalable Secure Multicasting”. Fourth International Symposium on Computer and Communications, July 1999.

[ECB80] Electronic Codebook Mode, The National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) Publication 81, "DES Modes of Operation," U.S. Department of Commerce, December 1980.

[EDW99] Edwards, W. K, “Core Jini”, Upper Saddle River, NJ: Prentice Hall, 1999.

[ERI94] Eriksson, H., “MBONE: The Multicast Bone”, Communications of the ACM, 37(8):54-60, August 1994.

[GAM95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., “Design Patterns Elements of Reusable Object-Orientated Software”, Addison-Wesley, 1995

[GOL99] Goland, Y., Cai, T., Leach P., Gu, Y., and Albright, S., “Simple Service Discovery Protocol,” IETF, Draft draft-cai-ssdp-v1-03, October 28 1999. http://search.ietf.org/internet-drafts/draft-cai-ssdp-v1-03.txt

[GON94] Gong, L., Shacham, N., "Elements of trusted multicasting". In Proc. IEEE Intl. Conf. on Network Protocols, pages 23-30, Boston, MA, USA, October 1994.

[GOV00] Govea, J., Barbeau, M., "Comparison of Bandwidth Usage: Service Location Protocol and Jini", September 2000.

[GUL96] Gulbrandsen, A., Vixie, P., "A DNS RR for Specifying the Location of Services (DNS SRV)," IETF RFC 2502, October 1996. http://www.rfc-editor.org/rfc/rfc2052.txt

[GUT99] Guttman, Erik, "Service Location Protocol : Automatic Discovery of IP Network Services,"IEEE Internet Computing, vol. 3, no. 4, pp. 71-80, 1999. http://computer.org/internet/

[GUT99a] Guttman, E., Perkins, C., Veizades, J., and Day, M., "Service Location Protocol, Version 2," IETF, RFC 2608, June 1999. http://www.rfc-editor.org/rfc/rfc2608.txt

[GUT99b] Guttman, E., "Service Location Protocol: Automatic Discovery of IP Network Services," IEEE Internet Computing, vol. 3, no. 4, 1999, pp. 71-80.

[GUT99c] Guttman, E., Perkins, C., Kempf, J., "Service Templates and Service: Schemes," IETF, RFC 2609, June 1999. http://www.rfc-editor.org/rfc/rfc2609.txt

[HAR97] Harney, H., Muckenhirn, C., "Group Key Management Protocol (GKMP) Architecture", RFC 2094, July 1997.

[HOD99] Hodes, Todd and Katz, Randy H., "A Document-based Framework for Internet Application Control," Second USENIX Symposium on Internet Technologies and Systems, Boulder, CO, 1999. http://daedalus.cs.berkeley.edu/publications/docu-usits99.ps.gz

[IBMa] IBM, "HTTP Server version 2.0". http://www-3.ibm.com/software/webservers/httpservers/

[IBMb] IBM, "WebSphere Application Server, Version 4.0". http://www-3.ibm.com/software/webservers/appserv/

[IBM02] IBM, "IBM WebSphere V4.0 Advanced Edition Handbook", Chapter 17, March 2002. http://www.redbooks.ibm.com/redpieces/pdfs/sg246176.pdf

[IPL] iPlanet, "iPlanet Web Server Enterprise Edition 6.0". http://docs.iplanet.com/docs/manuals/enterprise.html#60

[ITU97] ITU/ISO Recommendation X.500(08/97): Open Systems Interconnection – The Directory:Overview of concepts, models and services. International Telecommunication Union, 1997.

[JUR00] Jurafsky, D., Martin, J. H. Speech and Language Processing. Prentice Hall, 2000.

[LEE02] Lee, B.-S. and Bryant, Barrett R., "Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language," to appear in Proceedings of SAC 2002, the ACM Symposium on Applied Computing, 2002.

[LUQ01] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R., Kin, B. K., "DCAPS – Architecture for Distributed Computer Aided Prototyping System," Proceedings of RSP 2001, the 12th Rapid Systems Prototyping Workshop, 2001, pp. 103-108.

[MCG00] McGrath, R., " Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing", Presented at Center for Excellence in Space Data and Information Science NASA Goddard Space Flight Center, April 2000.

[MIC] Microsoft Corporation, "Internet Information Services V 5.0".http://www.microsoft.com/windows2000/server/evaluation/features/web.asp

[MIC98] Microsoft Corporation. DCOM Specifications, http://www.microsoft.com/oledev/olecom, 1998.

[MIT97] Mittra, S., "Iolus: A Framework for Scalable Secure Multicasting", In Proc. ACM SIGCOMM, pages 277-288, Cannes, France, September 1997.

[MOC87] Mockapetris, P., "Domain Names-Implementation and Specification," IETF RFC 1035, October 1987. http://www.rfc-editor.org/rfc/rfc1035.txt

[NIN02] Ninja, "The Ninja Project," http://ninja.cs.berkeley.edu, 2002.

[OMG00] Object Management Group, "Trading Object Service Specification," Object Management Group 2000. ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf.

[OMG01] Object Management Group,"Naming Service Specification" Object Management Group 2001.ftp://ftp.omg.org/pub/docs/formal/01-02-65.pdf

[ORF97] Orfali R, and Harkey, D. Client/Server Programming with JAVA and CORBA. John Wiley & Sons, Inc., 1997.

[PER99] Perkins, C., Guttman, E., "DHCP Options for Service Location Protocol," IETF RFC 2610, June 1999. http://www.rfc-editor.org/rfc/rfc2610.txt

[RAJ00] Raje, R. R., “UMM: Unified Meta-object Model for Open Distributed Systems”, Proceedings of ICA3PP 2000, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing", 2000, pp.454-465.

[RAJ01] Raje, R., Auguston, M., Bryant, B. R., Olson, A., Burt, C., “A Unified Approach for the Integration of Distributed Heterogeneous Software Components”, Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration, 2001, pp. 109-119.

[RAJ02] Raje, R., Auguston, M., Bryant, B. R., Olson, A., Burt, C., “A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components”, Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2002.

[RMI] Sun Microsystems, "Java Remote Method Invocation (RMI)", http://java.sun.com/products/jdk/1.2/guide/rmi/index.html

[RSA93] RSA Laboratories, "PKCS #5: Password-Based Encryption Standard," version 1.5, November 1993.

[SAL] Salutation Consortium, "Salutation," http://www.salutation.org

[SAL99a] Salutation Consortium, “Salutation Architecture Specification (Part-1 ) Version 2.1,” The Salutation Consortium 1999. http://www.salutation.org

[SAL99b] Salutation Consortium, “Salutation Architecture Specification (Part-2),” The Salutation Consortium 1999. http://www.salutation.org

[STA95] Stallings, W.,“Network and Internetwork Security”, Prentice-Hall Inc., 1995.

[SUNa] Sun Microsystems, “Designing Enterprise Applications with the J2EETM Platform”.http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/

[SUNb] Sun Microsystems, “Java 2 Platform, Standard Edition (J2SETM) v 1.4.0 Overview”. http://java.sun.com/j2se/1.4/

[SUN00] Sun Microsystems, “JavaTM Cryptography Extension 1.2.1 API Specification & Reference”, June 2000. http://java.sun.com/products/jce/doc/guide/API_users_guide.html

[SUN01a] Sun Microsystems, “Jini Architecture Specification, Version 1.2,” Sun Microsystems, December 2001, http://www.sun.com/jini/.

[SUN01b] Sun Microsystems, “Java$^{TM}$ 2 Platform Enterprise Edition Specification, Version 1.3”, Sun Microsystems, August 2001.
http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf

[SUN02] Sun Microsystems, “JavaTM Cryptography Architecture API Specification & Reference”, February 2002.
http://java.sun.com/j2se/1.4/docs/guide/security/CryptoSpec.html

[UDD00] uddi.org, “UDDI Technical White Paper”, September 2000,http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[UPNP99] Microsoft Corporation, “Universal Plug and Play Device Architecture Reference Specification,” Microsoft Corporation, November 10, 1999.
http://www.microsoft.com/hwdev/UPnP

[VAN65] van Wijngaarden, A. “Orthogonal Design and Description of a Formal Language”, Technical report, Mathematisch Centrum, Amsterdam, 1965.

[W3C] W3C, World Wide Web Consortium, “Leading the Web to its Full Potential”.
http://www.w3.org/

[WAH97] Wahl, M., Howes, T., Kille, S., “Lightweight Directory Access Protocol (v3),” IETF RFC 2251, December 1997. http://www.rfc-editor.org/rfc/rfc2251.txt

[WAL97] Wallner, D., Harder, E., Agee, R., “Key Management for Multicast: Issues and Architecture”, IETF Draft, July 1997.

[WEL] Welsh, Matt, "Ninja RMI: A Free Java RMI,"
http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html

[WIL99] Wilcox,M., ”Implementing LDAP”, Wrox Press Ltd.,1999

[YOU95] Young, D., “Object-Orientated Programming with C++ and OSF/Motif”, Prentice-Hall, 1995.

[ZIN95] Zinky, J.A., Bakken, D.E., and Schantz, R. Overview of Quality of Service for Distributed Objects. In Proceedings of the Fifth IEEE Dual Use Conference, 1995.